MTQR

Monomial Transformation Quadrature Rule

USER MANUAL

Installation, integration and execution

Politecnico di Torino

Dipartimento di Elettronica e Telecomunicazioni

Guido Lombardi, PhD,
guido.lombardi@polito.it
Davide Papapicco,
davide.papapicco@polito.it

Contents

Chapter 1

PRELIMINARIES

This user manual provides a detailed description of the functionalities, correct installation and usage of MTQR, an open-source and cross-platform C++ library for precise numerical integration. Users of the library can refer to the following content for building the static library, as well as integrating it in their own custom applications to generate specialised quadrature rules for integrand functions that can be modelled by sets of singular and generalised polynomials. The principles behind the implementation of the software are discussed in this first chapter, followed by instructions for compiling and linking the library in Windows and Linux (outlined in the second and third chapter) and finally by a concluding chapter that briefly describes the user interface. MTQR offers a high degree of flexibility concerning its usage and interaction, specifically:

- it provides high-precision quadrature rules (see Section 2.4) for multiple applications in numerical mathematics and scientific computing whenever functions that can be modelled by singular and generalised polynomials require to be integrated;
- it implements two *modes* of execution (namely the *loud* and *silent* modes) providing the options of both limiting the terminal's output information and output results (see Section 3.2) and accelerating and simplifying the overall workflow (see Section 3.3);
- it can be deployed as a supporting utility to generate, export and manipulate the new optimised samples and weights for a given set of singular and generalised polynomials (see Section 3.4).

Being a mathematical software, a solid grasp of various topics in mathematical and numerical analysis is an advantage for the user in order to fully exploit the library. In the second section of this chapter we provide a brief coverage on some of those concepts such as interpolatory quadrature formulae, Gaussian quadrature rules and asymptotic error estimation which are useful for a full comprehension of the proposed algorithm. Those are followed by the introduction of the monomial transformation, constituting the innovative core of the algorithm and the library, and its properties for quadrature of functions that can be modelled by singular and generalised polynomials. This overview is far from complete and self-contained, and we direct to [11, 12], and references therein, as the primary sources for a thorough theoretical exposition on the mathematical foundations of MTQR as well as on generalised polynomials (also known as **Müntz polynomials** , see section 1.2.4).

1.1 OVERVIEW

MTQR provides a framework for the high-precision numerical computation of definite integrals whose integrands are functions that can be modelled by singular and generalised polynomials, i.e. whose monomial terms have real degree strictly greater than -1. First and foremost one has to understand the motivation for such library to exist, in order to effectively decide whether or not its usage is necessary for her/his own application. Different techniques have been proposed to numerically approximate singular integrals however the accuracy of their performances is limited by their computational cost, effectively reducing their applications to canonical cases. Another motivation is the possibility to anticipate the required machine-precision in designing the quadrature for a given set of singular

and generalised polynomials avoiding to resort to iterative and adaptive techniques. We wrote MTQR with the aim of helping the user when both singular or generalised polynomials and high precision quadrature are a requirement thus avoiding to compromise between accuracy and time of execution. The pivot algorithm around which the library has been built (i.e. the monomial transformation) has its roots in the relative error estimation of the Gauss-Legendre (G-L) quadrature rule and on the ad-hoc numerical manipulation of the integration parameters (discussed in the following sections) according to the characteristics of the set of generalised polynomials to be integrated. The combination of those strategies results in an optimized **monomial transformation quadrature rule** that achieves precise numerical integration of singular, generalised polynomials. The background theory of the method refers to [11] while the work in [12] reports the automatised implementation and performance of the library. The monomial transformation quadrature rule is a result of the application of a novel exact asymptotic error estimate of the G-L quadrature rules introduced in [11] that shows the potentialities of such formulae to integrate generalized polynomials composed by monomials of real degree. These characteristics allows to define a target relative error for sets of generalised monomials (see IEEE floating-point formats' machine-epsilon in Table 1.1). Moreover, it enables the extension of the quadrature capability **(range of validity)** through the use of monomial transformations applied to arbitrary generalised polynomials and in particular to those characterised by an end-point singularity in the integration interval. We want to emphasize these features of MTQR right at the start of this user manual so that the user has a clear understanding of the contexts in which it becomes necessary. In fact, as shown in the test drivers (see Section 2.4), MTQR significantly outperforms the classical G-L quadrature rule for polynomials whose exponents' sequence is composed of either rational or irrational numbers (henceforth referred to as **Müntz sequence**, see Sub-Section 1.2.4). To consistently achieve such performance, our package implements a routine based on the following fundamental steps (discussed in more detail in the second chapter of this manual):

1. select a fixed threshold for the relative error of the integration rule (e.g. the machine-epsilon in IEEE double precision);
2. exploit the in-depth error estimation of the G-L quadrature to determine the lower and upper bound for the degrees associated to generalised monomials whose quadrature is within the prescribed relative error;
3. design the monomial transformation to generate a quadrature rule that obtains the specified relative precision on a selected family of generalized polynomials by exploiting the properties of error estimation.

Whilst the proposed software certainly allows for the integration of classical monomials and polynomials of integer degree, we remark that these integrands are not the cases for which the library was written nor its execution is necessary to achieve accurate results (interpolatory classical and generalised Gaussian rules do suffice). Indeed, the core aim of the library is the efficient and precise integration of generalised polynomials where other techniques may require heavy computational cost at run-time [5], or may result in specialized quadratures [13, 14]. MTQR is entirely written in C++17; beside the speed and versatility of the language, the primary motivation behind such choice was the accessibility to other open-source packages (see Section 2.1). These packages are essential for the implementation of different routines since they extract fundamental information for the precise quadrature of singular and generalised polynomial integrals. One such example is the ability of handling operations in *higher-than-double* floating-point arithmetic (e.g. IEEE quadruple floating-point format or higher, see Table 1.1 below), provided in the **Boost's Multiprecision** library, without which the precise computation of the quadrature parameters and the quadrature itself, for the most computationally demanding cases, would have not been guaranteed. MTQR implements methods that suit a very specific demand in numerical analysis so we used a procedural/modular approach in its writing (as opposed to an object-oriented paradigm) in which all the different functions are hidden behind the library's access point `mtqr` module (see Section 2.1 and 2.2).

Floating-point formats			
Common name (official)	**single (binary32)**	**double (binary64)**	**quadruple (binary128)**
n. bits	32	64	128
n. decimal digits	7	16	34
epsilon	1.1921 e-07	2.2204 e-16	1.9259 e-34
real min	1.1755 e-38	2.2251 e-308	3.362 e-4932
real max	3.4028 e+38	1.7977 e+308	1.190 e+4932

Table 1.1: Some parameters of the most common floating-point formats specified by the IEEE 754 standard [10].

1.2 MATHEMATICAL BACKGROUND

Given a function $f : (a, b) \to \mathbb{R}$, which approximates a target arbitrary integrand with a set of singular and generalised polynomials, we consider its definite integral over the generic integration interval $(a, b) \subset \mathbb{R}$

$$I(f) := \int_a^b f(x)dx \tag{1.1}$$

The driving purpose of MTQR is to compute a precise numerical approximation of I(f). It is customary to refer to polynomials characterised by natural integer degree $k \in \mathbb{N}$, however our library purposely extends such definition to work with the larger sub-space of generalised polynomials of non-integer (real) degree, i.e. $f \in \mathbb{P}_\alpha[x]$, $\alpha \in (-1, +\infty)$. Of course the analytic integration of polynomial functions is trivial, always leading to an exact form (polynomial) of the primitive $\tilde{f}$ to be evaluated on the integration bounds

$$I(f) = \left[\tilde{f}(x)\right]_{x=a}^{x=b}, \quad \tilde{f} \in \mathbb{P}_{\alpha+1}[x], \ \forall f \in \mathbb{P}_\alpha[x] \tag{1.2}$$

However there are several cases in numerical and computational mathematics in which those integrals are required to be calculated numerically; one such instance are Galerkin methods of approximation of partial differential equations (PDEs) where the local weak formulation of the model requires the numerical evaluation of polynomial integrands modelling some products of the elemental basis functions (e.g. in the Finite and Boundary Element Methods). In famously distinguished applications in scientific computing (e.g. in computational electromagnetics, CEM) non-classical generalised polynomials are used in order to model basis functions featuring an endpoint singularity in either or both the bounds of integration, which cannot be represented exactly or with sufficient accuracy when modelled through regular high-order polynomials.

1.2.1 INTERPOLATORY QUADRATURE

In order to introduce the topic, we let $f : (a, b) \to \mathbb{R}$ to be any sufficiently regular function; we want to compute $I(f)$ although unfortunately its primitive is not known. One solution is to approximate the function with a polynomial interpolating some of its values at sampled point in (a, b), called **nodes**. Such nodes constitute a finite, countable set $\mathcal{I}_h := \{x_1 , x_2 , \dots , x_n\}$ which can be thought as a discretisation of the original domain of integration (a, b). The cardinality of such set is n and it is linked to the degree of the polynomial interpolating $f(x)$; in particular such degree is $n - 1$ (i.e. the degree is always equal to the number of nodes minus 1). The interpolating polynomial $\mathbb{P}_{n-1} \ni \mathcal{L}_{n-1}(x) := \sum_{j=1}^{n} \ell_j(x) f(x_j)$ is expressed in the **Lagrangian basis** whose generators $\ell_j(x) \in \mathbb{P}_{n-1}$ are also polynomials of the same $n - 1$ degree. We will not address the theoretical aspects of Lagrangian interpolation and approximation theory since the user may refer to the classical literature about the topic. We can now express the integrand through an approximating polynomial of degree $n - 1$

$$f(x) = \mathcal{L}_{n-1}(x) + E_n^{'}(f) = \sum_{j=1}^{n} \ell_j(x) f(x_j) + E_n^{'}(f) \quad x_j \in \mathcal{I}_h \, , \ \forall j = 1, ..., n \tag{1.3}$$

By substituting (1.3) in (1.1) we obtain an approximation of the initial integral known as an **interpolatory quadrature formula**

$$I(f) = \int_a^b (\mathcal{L}_{n-1}(x) + E_n^{'}(f))\, dx = \sum_{j=1}^{n} f(x_j) \int_a^b \ell_j(x)\, dx + \int_a^b E_n^{'}(x) = \sum_{j=1}^{n} w_j\, f(x_j) + E_n(f) \tag{1.4}$$

where we define the **weights** w_j of the quadrature formula as the definite integral of the Lagrangian basis' generators and the **remainder** as $E_n(f) := \int_a^b E_n^{'}(f)\, dx$. It is a well known result in numerical analysis that any interpolatory quadrature rule made on n samples is exact, i.e. $E_n(f) = 0$, if the integrand is any polynomial of degree up to $n - 1$.

1.2.2 GAUSSIAN QUADRATURE FORMULAE

One of the main issues with numerical integration using interpolatory quadrature is the selection of the nodes' distribution. The simplest case is the uniformly distributed partition $\mathcal{I}_h = \left\{x_j = a + (j - 1)\, h \, , \ h := \frac{b-a}{n}\right\}_{j=1,\dots,n}$ which results in the **Newton-Cotes quadrature formula**. As reported in the classical literature, better choices for such distribution, depending on the properties and behaviour of $f(x)$, are available. In particular the **Gaussian quadrature formula** is one where each of the nodes uniquely corresponds to the root of an orthogonal polynomial. To introduce these approximations we now let the integrand to be an arbitrarily regular function

$$\mathcal{C}^m[\mathcal{I}] \ni f(x) = w(x)g(x)\,, \quad m \in \mathbb{N} \tag{1.5}$$

where the factorised $g(x)$ always contains its most regular part whereas $w(x)$, known as the **weight function** contains, if present, any irregular and/or singular part of $f(x)$. If the weight function is not null everywhere in (a, b) and we can find an infinite sequence of classical polynomials $P := \left\{p_j(x) \in \mathbb{P}_j\right\}_{j=0,1,\ldots}$ s.t.

$$\int_a^b w(x)p_j(x)p_k(x)\,dx = \alpha_{jk}\delta_{jk}\,, \quad \forall p_j\,,\ p_k \in P \tag{1.6}$$

where δ_{jk} is the Kronecker delta, then P represents a **system of orthogonal polynomials** w.r.t. the specified weight function $w(x)$. Any Gaussian quadrature formula built on n nodes is exact for any classical polynomial integrand $g(x)$ of degree up to $2n-1$. It is therefore easy to see why Gaussian quadrature rules are usually preferred over Newton-Cotes formulae when dealing with regular functions with a known, factorised weight $w(x)$. Although Gaussian quadrature allows the exact integration with respect to a limited number of weight functions, throughout the years they have been generalised using refined approximations and numerical efforts [5]. The simplest and arguably most widely used Gaussian quadrature rule is the Gauss-Legendre (G-L) formula, for which the sequence P is made of **Legendre's polynomials** that are orthogonal w.r.t. the constant weight function $w(x) = 1$ over the interval $(a = -1\,,\ b = +1)$.

1.2.3 ASYMPTOTIC ERROR ESTIMATION

This subsection analyzes the accuracy of the G-L quadrature formula with n nodes. As reported in the previous sub-sections, if $f(x) \in \mathbb{P}_k\,,\ k \in \mathbb{N}$ we'd have $E_n(f) = 0,\ \forall k \leq 2n-1$. In analysing the performance of a numerical algorithm it is often more appropriate to define and use an a-posteriori (actual) relative error associated to the approximating techniques, rather than an *absolute* one s.a. the remainder $E_n(f)$. For the specific case of the numerical integration of an arbitrary function $f(x)$, via a G-L formula, we have

$$R_n^{(a)}(f) = \frac{|I(f) - I_n(f)|}{|I(f)|}\,, \quad I_n(f) := \sum_{j=1}^{n} w_j\, f(x_j) \tag{1.7}$$

where the sequence of pairs $\{(x_j, w_j)\}_{j=1,\ldots,n}$ is the set of nodes and weights of the G-L quadrature rule. In those cases, where the performance of the quadrature rule, in terms of the associated relative error, needs to be known a priori, we must refer to an estimate of (1.7)

$$R_n(f) = \frac{|E_n(f)|}{|I(f)|} \sim R_n^{(a)} \tag{1.8}$$

where $E_n(f)$ is the estimated reminder of the quadrature rule. In [11], one of the authors of the present work, derived a closed form a-priori error asymptotic estimation of $E_n(f)$ for a G-L formula applied to a generalised monomial term $f(x) = x^\lambda$ with real-valued degree $\lambda > -1$.

$$E_n(x^\lambda) = -2^{-2\lambda}\,\lambda\,\sin(\pi\lambda)\left(\frac{B(2\lambda, 2n-\lambda)}{2n+\lambda} - \frac{B(2\lambda, 2+2n-\lambda)}{2+2n+\lambda}\right) \tag{1.9}$$

where $B(z, w) := \frac{\Gamma(z)\Gamma(w)}{\Gamma(z+w)}$ is the Euler's Beta function. This result is important for the development of an ad-hoc quadrature rule whose error can be computed a-priori for any generalised polynomial integrand function (see Sub-Section 3.3 in [12] and references therein). In particular, given a specified number of quadrature nodes n, we can clearly see in Figure 1.1 how the a-priori relative error estimate $R_n(x^\lambda)$ behaves asymptotically with λ. For instance we immediately notice that the value of $R_n(f)$, with a fixed n samples, is below a pre-determined arbitrary precision (in terms of relative error) within a specific, finite range of values of λ which we identify as $(\lambda_{min}, \lambda_{max}) \subset (-1, +\infty)$. In practice we can specify a finite-arithmetic threshold, for example the machine-epsilon in double f.p. format as reported in the figure below (see constant black line). Once identified, the interval $(\lambda_{\min}, \lambda_{\max})$ exactly coincides with the real range of exponents of the set of generalised polynomial functions whose numerical integration (using the specified G-L quadrature rule) would be exact in finite arithmetic.

1.2.4 MÜNTZ THEOREM AND GENERALISED POLYNOMIALS

In the opening subsection we mentioned that MTQR extends the integration capabilities of the G-L formulae to generalised polynomials of real degree in the range of exponents $(-1, +\infty)$ by means of monomial transformations. As the reader may find in [11] and references therein, a complete theory on the properties of these kind of functions relates to Müntz polynomials, orthogonal Müntz polynomials and their numerical computation.

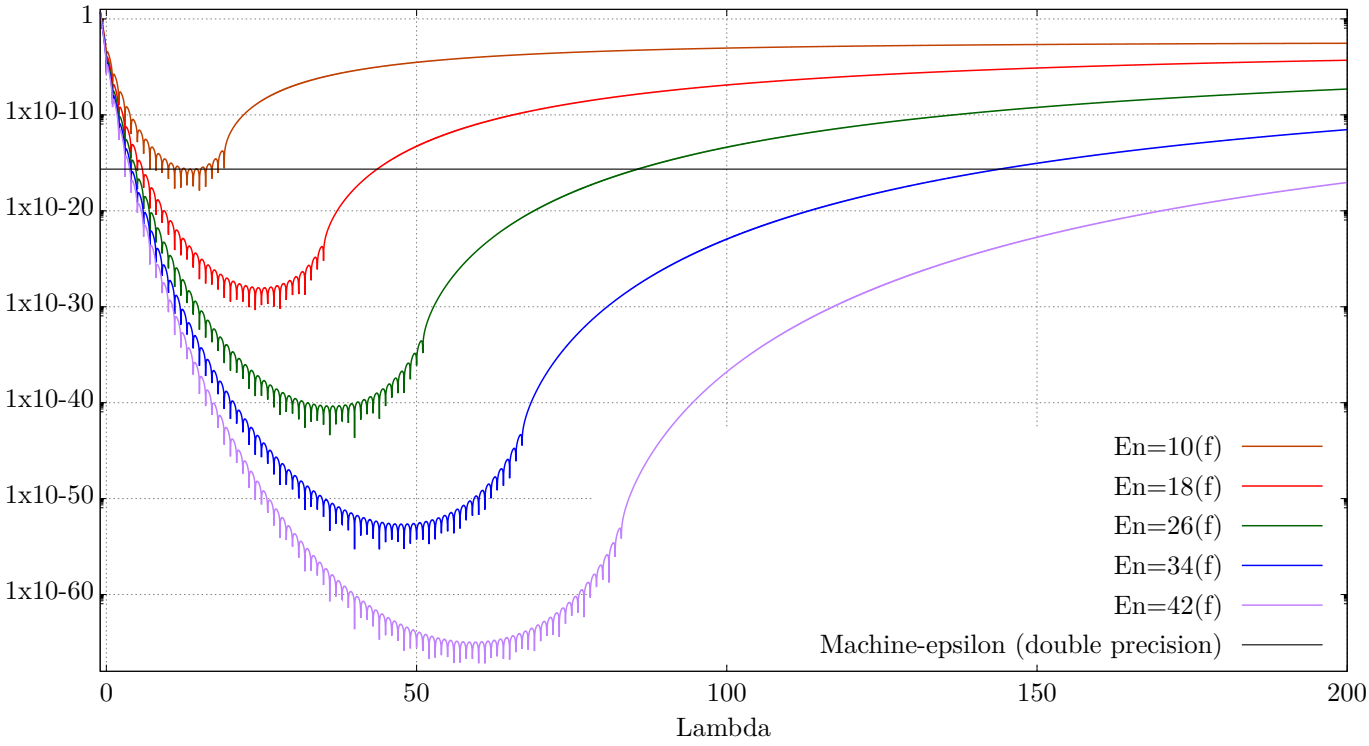

Figure 1.1: Evaluation of the relative error estimate $R_n(x^\lambda)$ (in logarithmic scale) provided in (1.8) for $n = 10, 18, 26, 34, 42$. Notice that for a fixed n we have that $R_n(x^\lambda)$ is smaller than an arbitrary precision threshold (solid horizontal black line) in $(\lambda_{min}, \lambda_{max})$. Furthermore, as long as λ is a natural integer value smaller than $2n-1$, the relative error shows negative spikes, therefore mimicking the property of a G-L quadrature to feature zero reminder. Finally it is important to emphasize that for $n = 10$ the estimate barely reaches the machine-epsilon threshold for double precision entailing that it is the minimum number of nodes that can be used to achieve double precision quadrature for a limited range of real λ.

We assume that the generalized polynomials feature a sequence of (real) exponents that is a subset of a Müntz sequence, i.e. an ordered sequence of real numbers $\Lambda := \{\cdots > \lambda_j > \lambda_{j-1} > \cdots > \lambda_1 > -1\}$ s.t.

$$\sum_{j=1}^{+\infty} \frac{1}{\lambda_j} = +\infty \tag{1.10}$$

Such condition satisfies the known **Müntz theorem** for which the vector space $\Pi(\Lambda) := \text{span}\{x^{\lambda_j}\}$ spanned by the monomials of degree in Λ is dense in $\mathcal{C}(0,1]$. Elements $p^{(m)}(x) = \sum_{j=1}^{r} c_j\, x^{\lambda_j} \in \Pi(\Lambda)$ are therefore known as **Müntz polynomials** of degree λ_r defined in $(0,1]$ where $\Lambda_r := \{\lambda_r > \cdots > \lambda_j > \lambda_{j-1} > \cdots > \lambda_1 > -1\}$, $r \in \mathbb{N}_0$. In the following we shall assume that any polynomial with r terms having at least one constituting monomial of non-integer degree are to be considered Müntz (generalised) polynomials. The monomial transformation quadrature rule implemented in MTQR is capable of approximating the integral of functions modelled by generalised polynomials defined in $(0,1)$ and, crucially, featuring an endpoint singularity at $x = 0$. The proposed quadrature allows to obtain an arbitrary fixed precision for the integral combining the G-L quadrature's performance (see in Figure 1.1 the relative error falls below the machine-epsilon threshold in double f.p. format) with an ad-hoc designed monomial transformation by exploiting the properties of error estimation.

1.2.5 INTEGRATION INTERVAL

We consider the problem of numerically integrating a function that is approximated appropriately with a set of generalised polynomials in the interval (a,b). This problem can always be reformulated in $(0,1)$ with a potential (endpoint) singularity in $x = 0$ via affine trasformation, i.e.

$$I(f) = \int_0^1 f(\tilde{x}) d\tilde{x}, \tag{1.11}$$

This choice is due to:

- uniquely investigate the precision property of G-L quadrature and derive unique rules to design a monomial transformation for the numerical integration algorithm;
- avoid or at least severely restrict the effects of numerical cancellation in the numerical quadrature of singular generalised polynomials with endpoint singularities other than $x = 0$;
- to be consistent with the definition of Müntz polynomials and approximation theory.

The second property is necessary to avoid rounding errors in the calculation of the transformed quadrature nodes. In fact, as the quadrature points tend to be clustered around the singularity, when it is located far from the origin $x = 0$, the resulting samples may be affected by numerical cancellation. Often, in practical problems, integration on an arbitrary interval (a, b) is requested. For this reason, we need to define an affine transformation to map the original, generic, integration interval (a, b) into $(0, 1)$. To that purpose, let us introduce a (linear) affine map $\varphi : (a, b) \to (0, 1)$ specified as

$$\varphi(y) := \alpha\, y + \beta = \tilde{x}\,, \quad y \in (a, b) \tag{1.12}$$

It is easy to see that $\varphi(y = 0) = \beta \equiv a$ and $\varphi(y = 1) = \alpha - \beta \equiv b$. We obtain

$$I(f) = \int_a^b f(y)dy = \int_0^1 f\Big((b-a)\tilde{x} + a\Big)\, J_{[a,b]}\, d\tilde{x}\,, \quad J_{[a,b]} := b - a \tag{1.13}$$

This new form (1.13) of the integral can now be computed numerically using the monomial transformation quadrature rule by considering that singular integrands at the endpoint $y = a$ will now have a singularity in $\tilde{x} = 0$. The same transformation can be useful to easily map standard G-L quadrature samples and weights $\{y_j, v_j\}_{j=1,...,n}$ defined in $(-1, 1)$ to the less common G-L quadrature samples and weights $\{\tilde{x}_j, \tilde{w}_j\}_{j=1,...,n}$ defined in $(0, 1)$ which is considered as the starting point to apply the monomial transformation quadrature rule. In this specific case we obtain

$$I(f) = \int_{-1}^1 f(y)dy = \sum_{j=1}^n f(y_j)v_j = \int_0^1 f\Big(2\tilde{x} - 1\Big)\, 2\, d\tilde{x} = \sum_{j=1}^n 2\,\tilde{w}_j f(2\tilde{x}_i - 1) \tag{1.14}$$

thus

$$\tilde{w}_j = \frac{v_j}{2}, \;\; \tilde{x}_j = \frac{y_j + 1}{2}, \quad j = 1, ..., n \tag{1.15}$$

One important remark is that, once the new sample and weights $\{\tilde{x}_j, \tilde{w}_j\}$ have been derived for the proposed monomial transformation quadrature (the algorithm is described in the following Sub-Section 1.2.6), we should avoid to map them back to the original integration interval (a, b) due to numerical cancellation in the representation of the samples near the singular endpoint, say $y = a$. The novel quadrature is therefore safely applied in $(0, 1)$ by re-formulating in it the original integration problem defined in (a, b) as done in (1.13).

1.2.6 MONOMIAL TRANSFORMATION

Given a set of Müntz (generalised) polynomials we define $\lambda_{\min}$ and $\lambda_{\max}$ as the smallest and greatest values, respectively, in the Müntz sequence of exponents. From Sub-Section 1.2.4 we know that $\lambda_{\min} > -1$ (we consider only integrable functions) and refer to $\lambda_{\max}$ as the largest degree of the polynomials. Recall also that only for classical polynomials (i.e. $\lambda_j \in \mathbb{N}$) we obtain $E_n(f) = 0$ when using G-L formulae with $n = \lceil \frac{\lambda_{\max}+1}{2} \rceil$ samples. Regardless, generalised singular polynomials, s.a. those in Müntz vector space, are integrated with poor accuracy by classical G-L with a specified number of nodes when containing generalised monomials with integration relative errors beyond the selected threshold. Analysis of (1.8) and (1.9) allow to fully exploit the performance of G-L quadrature rule when applied to generalised monomials; specifically they are used to design an ad-hoc monomial transformation $\gamma(\tilde{x}) : (0, 1) \to (0, 1)$, for the set of polynomials identified by $\lambda_{\min}$ and $\lambda_{\max}$

$$x = \gamma(\tilde{x}) = \tilde{x}^r\,, \quad \tilde{x} \in (0, 1) \tag{1.16}$$

with the constraint (see derivation in [11, 12])

$$\frac{1 + \beta_{\min}(n)}{1 + \lambda_{\min}} < r < \frac{1 + \beta_{\max}(n)}{1 + \lambda_{\max}} \tag{1.17}$$

where $\beta_{\min}(n)$ and $\beta_{\max}(n)$ are the minimum and maximum exponents for generalised monomial functions that can be integrated with a relative error strictly smaller than a fixed finite precision using G-L quadrature with n samples,

see also Figure 1.1. Given some expression for $(\beta_{\min}(n), \beta_{\max}(n))$ enforcing (1.17) yields a unique minimum value of n (henceforth referred to as n_{min}) that satisfies the above constraint of being the (sole) real root of

$$(-4.0693 \cdot 10^{-3} + 4.1296 \cdot 10^{-4} n)[(8.8147 + 1.0123 \cdot 10^{-1} n^2) \cdot (1 + \lambda_{\min}) - (1 + \lambda_{\max})]^3 - (1 + \lambda_{\max})^3 = 0 \quad (1.18)$$

Given n_{min} one can easily compute r, for example, as the midpoint between the upper and lower bounds of the inequality in (1.17) and easily derive new nodes and weights computed, according to [11, 12], via (1.16) and

$$x_j = \tilde{x}_j^r, \quad w_j = r \tilde{x}_j^{r-1} \tilde{w}_j \quad \forall j = 1, ..., n \quad (1.19)$$

respectively, where $\tilde{x}_j$, $\tilde{w}_j$ are the classical G-L nodes and weights mapped in $(0, 1)$. The transformed new nodes and weights are optimised for the accurate numerical integration in $(0, 1)$ of the two monomials with the minimum and maximum exponents, i.e $c_{min}\, x^{\lambda_{min}}$ and $c_{max}\, x^{\lambda_{max}}$, guaranteeing the same f.p. precision for any additional monomial term $c\, x^{\lambda}$ with $\lambda \in (\lambda_{min}, \lambda_{max})$. For example, consider the following Müntz polynomial

$$\Pi(\Lambda) \ni p(x) = e x^{e+\frac{1}{4}} - \frac{1}{10} x^{-\frac{1}{2}} + \frac{1}{10} x^{-\frac{\pi}{4}}, \quad \Lambda = \left\{ -\frac{\pi}{4}, -\frac{1}{2}, e + \frac{1}{3} \right\} \quad (1.20)$$

A visual representation of $p(x)$ is reported in Figure 1.2 together with the effect of the ad-hoc designed monomial transformation on the quadrature for double precision integration. The Table 1.2 below lists the classical G-L nodes and weights alongside their (1.16) and (1.19) respective counterparts used by the proposed monomial transformation quadrature rule. In this case, the library only uses $n_{min} = 14$ quadrature samples for a monomial transformation order $r = 28.77$.

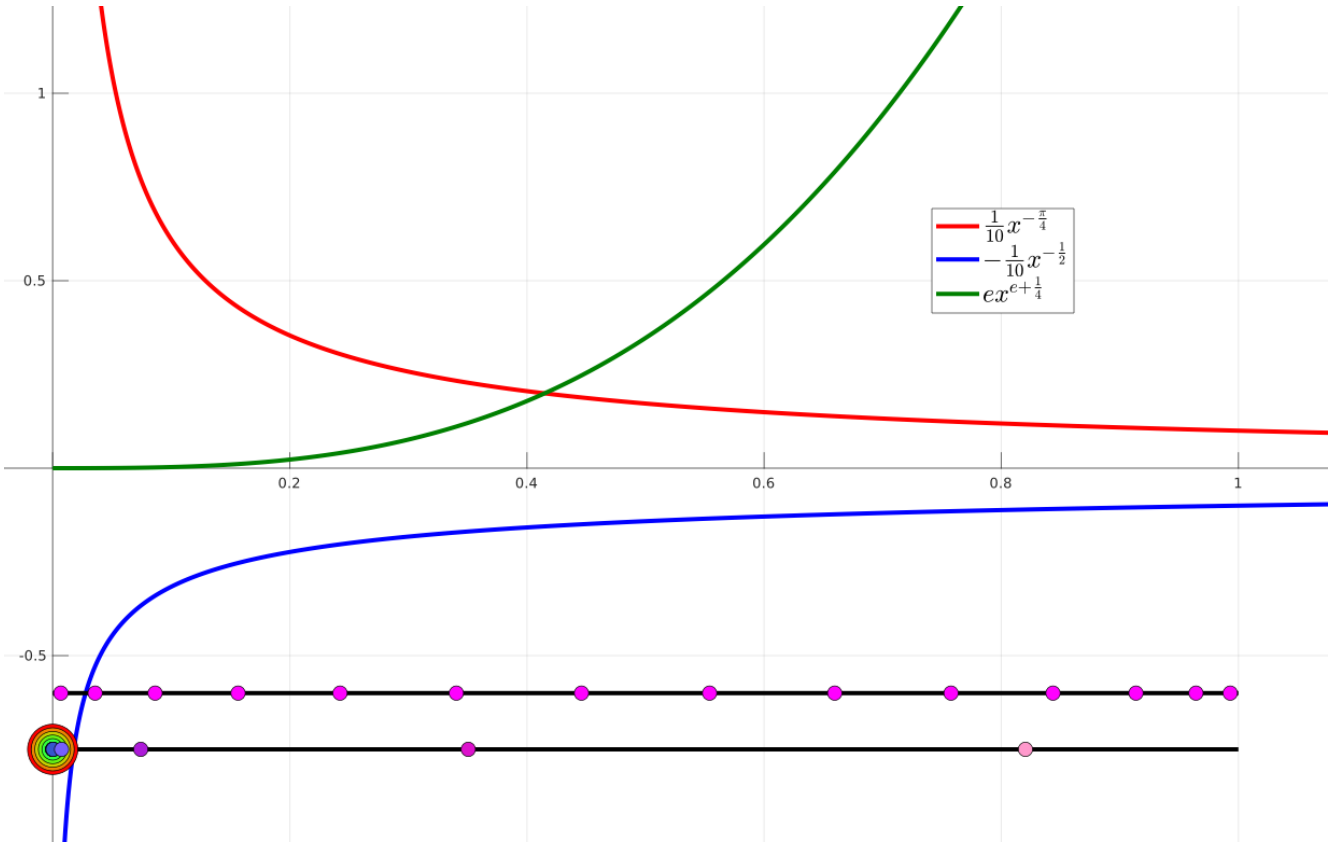

Figure 1.2: Plot of the monomial terms in (1.20). On the lower part of the graph a distribution in $(0, 1)$ of the $n = 14$ nodes of a classical G-L formula (magenta) and the new samples obtained by the monomial transformation quadrature rule (linear color gradient) is reported (with double precision threshold for integration). Due to the clustering of the latter (see Table 1.2), they are plotted as scaled-down circles as we move from $a = 0$ to $b = 1$. Observe how classical G-L nodes in $(0, 1)$ are inefficiently organised to capture the singularity in $x = 0$. Conversely the action (1.16) (whose order $r = 28.77$ is given by (1.17)) re-arranges the vast majority of nodes around $x = 0$ which is an endpoint singularity for the generalised monomial terms $x^{-\frac{\pi}{4}}$ and $x^{-\frac{1}{2}}$ and an irregular endpoint for the generalised monomial term $x^{e+\frac{1}{4}}$.

	Classical G-L samples		**MTQR samples**	
$j \in \mathbb{N}$	$\tilde{x}_j \in (0,1)$	$\tilde{w}_j \in \mathbb{R}^+$	$x_j \in (0,1)$	$w_j \in \mathbb{R}^+$
1	0.00685809565159384	0.0175597301658759	5.58247922741512e-63	4.11231619931278e-61
2	0.0357825581682132	0.0400790435798801	2.44695140063495e-42	7.88526679349951e-41
3	0.0863993424651175	0.0607592853439516	2.52951532315787e-31	5.11781542764354e-30
4	0.156353547594157	0.0786015835790968	6.51929796325047e-24	9.42908328714637e-23
5	0.242375681820923	0.0927691987389689	1.95675667095192e-18	2.15474891008643e-17
6	0.340443815536055	0.102599231860648	3.44180025480158e-14	2.98421017862521e-13
7	0.445972525646328	0.107631926731579	8.13715952769522e-11	5.65003223119147e-10
8	0.554027474353672	0.107631926731579	4.18125660540031e-08	2.33701617760019e-07
9	0.659556184463945	0.102599231860648	6.30683776156361e-06	2.82259817558914e-05
10	0.757624318179077	0.0927691987389689	0.000340288421120315	0.00119878736245905
11	0.843646452405843	0.0786015835790968	0.00750950989496675	0.0201292451904070
12	0.913600657534883	0.0607592853439516	0.0742930434150393	0.142150858859985
13	0.964217441831787	0.0400790435798801	0.350516762468882	0.419175839329777
14	0.993141904348406	0.0175597301658759	0.820378484398468	0.417316809008697

Table 1.2: List of all the $n = 14$ samples (nodes and weights) of G-L quadrature in $(0, 1)$ and of the monomial transformation quadrature rule with $r = 28.77$ as depicted in Figure 1.2.

Chapter 2

INSTALLATION

In the following chapter we illustrate how to build executable applications with MTQR as well as the dependencies needed at link-time. We begin from the latter topic by outlining the third-party packages necessary for the library prior to the compilation itself. Later, a brief description of the organisation and structure of the code is given in order to facilitate the comprehension of the user interface. Then, we explain how to build the library itself and finally illustrate its core features by running testing applications shipped with the software. We postpone to the next chapter 3 a description of the two *modes* of execution, namely the **loud** and **silent mode** .

2.1 DEPENDENCIES

As reported in the opening section, MTQR is entirely written in C++17; besides the speed and versatility of the language, the primary motivation behind such choice was the accessibility to other open-source mathematical packages. Indeed MTQR relies on two libraries that are well-known in the scientific computing and open-source communities listed below:

- **Boost C++ libraries** [2]: a vast, peer-reviewed, collection of mostly header source files. In particular MTQR extensively uses the `Multiprecision` library, where non-native higher precision f.p. formats have been implemented as C++ data-types. In particular the library depends on the `cpp_bin_float` back-end, and more specifically the `cpp_bin_float_quad` number type (henceforth referred to as `float128`) is used as a drop-in replacement of C++ f.p. native data types to efficiently implement IEEE 754 quadruple precision across multiple architectures and compilers.

- **GSL - GNU Scientific Library** [6]: required by MTQR in only one instance, that is the computation of the roots of (1.18) via the method `gsl_poly_complex_solve` . Alternative solvers and root-finders are available however in our case we were also interested in automatically locating the sole real root of such polynomial (corresponding to $n_{\min}$ in (1.18)), hence the choice we made.

The user must make sure that both libraries are correctly installed on the local system and that they are either placed in the default search paths of the user's preferred C++ compiler and linker or that the appropriate environment variable is amended according to the desired location of the source code of the aforementioned libraries. Platform-specific instructions are outlined in the following subsections for correctly installing and placing both the aforementioned dependencies as well as the building tools (**CMake** and **GNU make**) needed to correctly compile MTQR in Windows and Linux. We remark that the installation and build of the library and all its dependencies has been tested on multiple combinations of OS and C++ compilers as reported in Table 2.1.

Platform combinations							
Env. n.	OS	Processor	Compiler	Make	CMake	Boost	GSL
1	Ubuntu 22.04.1	Intel Core i5-6300U @ 3.0 GHz	GCC 11.3.0	4.3.0	3.21.1	1.74	2.7
2	Ubuntu 20.04.3	Intel Core i5-1035G1 @ 1.0 GHz	GCC 9.3.0	4.2.1	3.20.2	1.77	2.5
3	CentOS 8.5	Intel Core i7-10700 @ 2.9 GHz	GCC 8.5.0	4.2.1	3.20.4	1.66	2.5
4	Windows 11	Intel Core i7-10700 @ 2.9 GHz	MVSC 14.32	N/A	3.25.4	1.81	2.7
5	Windows 11	AMD Ryzen 7 5875U @ 2.0 GHz	MVSC 14.34	N/A	3.25.2	1.81	2.7

Table 2.1: Tested development environments.

2.1.1 LINUX

The proper installation of all the dependencies required by MTQR in Linux is straightforward thanks to package managers s.a. the **advanced package tool** or **yellowdog updater modified** (based on the user's distro). The user shall run the following commands on the terminal and verify the correct installation by checking their version[1].

Installation of third-party libraries (Linux)

```
# For Debian-based distros (Ubuntu, Mint, Knoppix, Kali ...)
user@machine: home> sudo apt-get update
user@machine: home> sudo apt-get install build-essential libboost-all-dev libgsl-dev cmake
user@machine: home> gcc --version
user@machine: home> make --version
user@machine: home> cmake --version
# For RPM-based distros (CentOS, Fedora, SUSE, Scientific Linux ...)
user@machine: home> sudo dnf makecache --refresh
user@machine: home> sudo dnf install http://repo.okay.com.mx/centos/8/x86_64/release/okay-
    ↪ release-1-1.noarch.rpm
user@machine: home> sudo dnf install gcc gcc-c++ make cmake
user@machine: home> sudo dnf install boost gsl libquadmath
user@machine: home> sudo dnf install boost-devel gsl-devel
user@machine: home> gcc --version
user@machine: home> make --version
user@machine: home> cmake --version
```

Notice that in the above lists of terminal commands we installed both **CMake** and **make**. This is because, while the preferred and de-facto standard building tools used to compile C++ libraries is `cmake`, on Linux we decide to allow the users to have the additional option of building MTQR also using only `make`. For more information on how MTQR allows using Make for compiling the library we redirect on subsection 2.3.1.

2.1.2 WINDOWS

Installation of the dependencies in Windows is somewhat more cumbersome when compared to Linux; we hereby report the procedure we tested for the simplest and most secure installation of the third-party packages in order to achieve a successful build of MTQR:

- GSL can be installed via the **vcpkg** package manager. The latter is downloaded on the local machine by using **Windows PowerShell**; we recommend the root directory as installation path for the package manager as this facilitates the setup for the user since the root directory is searched by default by the vast majority of C++ compilers and build systems available on Windows, including MSBuild, Clang, Ninja and MSVC. A list of terminal commands to perform a correct installation and set up for these dependencies is reported below;
- one of the most practical features of Boost's libraries is that they are (mostly) made by header files which do not need to be compiled; MTQR's dependency on Boost is restricted to the Multiprecision library. The user shall download the archived `boost_1_81_0.7z` file from the official Boost v1.81.0 release ([3]), decompress it in a folder named `boost` and place it in the `C:\` root directory. This way the user is sure that `CMake` can find Boost headers and properly include them in the MTQR's source code;

[1] CentOS Linux 8 had reached the End Of Life at the end of 2021. For this reason, to update CentOS 8, you need to change the mirrors to vault.centos.org where they will be archived permanently. This procedure is described at https://techglimpse.com/failed-metadata-repo-appstream-centos-8. CentOS Linux 9 requires the installation of gsl-devel via sudo dnf install http://mirror.stream.centos.org/9-stream/CRB/x86_64/os/Packages/gsl-devel-2.6-7.el9.x86_64.rpm.

- on Windows MTQR can only be compiled using **CMake**. The easiest and safest procedure of installing CMake on Windows is by downloading the binaries from the original webpage [4] and running the installer.

Installation of third-party libraries (Windows)

```
PS C: > git clone https://github.com/Microsoft/vcpkg.git
PS C: > cd .\vcpkg\
PS C: > .\bootstrap-vcpkg.bat
PS C: > .\vcpkg.exe integrate install
PS C: > .\vcpkg.exe install gsl gsl:x64-windows
PS C: > .\vcpkg.exe list
PS C: > cmake --version
```

In our build we used Microsoft Visual Studio 2022 which automatically runs CMake in any directory containing a CMakeLists.txt and, if successful, autonomously creates the associate project files (see subsection 2.3.2).

2.1.3 INTEGRATION

The dependencies are constituted of large libraries, although MTQR uses a limited amount of the methods they provide. We therefore made sure that only the necessary parts of the libraries are included in the source code, striving to maintain a clean and light final product. This is evidenced in the content of the library's principal header file mtqr.h which is reported in the snippet below.

mtqr.h

```
#ifndef MTQR_H
#define MTQR_H

#include <iostream>
#include <algorithm>
#include <iomanip>
#include <string>
#include <vector>
#include <tuple>
#include <fstream>
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <boost/math/constants/constants.hpp>
#include <boost/multiprecision/cpp_bin_float.hpp>
#include <gsl/gsl_poly.h>

namespace boomp = boost::multiprecision;
typedef boomp::cpp_bin_float_quad float128; // quadruple precision f.p. format

#include "vector_operations.h"  // includes header file for vector data structures and
    ↪ operations
#include "data_management.h"  // includes header file for data handling and management
#include "monomial_transformation.h" // includes header file for functions computing the
    ↪ monomial transformation

#define EPS std::numeric_limits<double>::epsilon() // sets double machine-epsilon as
    ↪ treshold
#define PI boost::math::constants::pi<float128>() // defines pi with 34 decimal digits
#define E boost::math::constants::e<float128>() // defines e with 34 decimal digits

// Primary module's (polymorphic) declaration
// Loud mode
template<typename T>
void mtqr(std::vector<T>& muntz_sequence, std::vector<T>& coeff_sequence);
// Silent mode
std::vector<std::vector<double>> mtqr(double lambda_min, double lambda_max);

#endif // MTQR_H
```

In the above header we find the definitions of some important constants that are used throughout the library preceding the declaration of `mtqr`, the method that exposes MTQR's library to the outer environment (i.e. any user's source code invoking it). We will henceforth refer to `mtqr` as the primary module of the library. The user may notice that we overloaded the primary module by taking different inputs and returning different outputs depending on whether the loud or silent mode is executed. We observe that compile-time polymorphism in C++ is not allowed when only the return type of the function differs, resulting in identical signatures for the symbol.

2.2 STRUCTURE

As stated above, the primary module constitutes the access point for the user to the content of MTQR. The library codebase is relatively compact and simply organised in a total of 12 methods, which are declared in 3 different header files and whose implementation/definition is found in their respective source files listed below:

- `monomial_transformation.cpp` contains every method associated with the computation of the monomial transformation quadrature rule, ranging from the monomial map itself (i.e. $\beta_{\min/\max}$ and r) to the quadrature parameters (i.e. $\tilde{x}_j$, $\tilde{w}_j$, $J_{[a,b]}$, etc...). To provide an easier reference for code debugging and amendment, a naming scheme of these methods is adopted. Every function in this file is in fact named **compute<NameOfFunction>** as it can be evinced from the corresponding header file containing such functions' declarations

monomial_transformation.h

```
#ifndef MON_TRF_H
#define MON_TRF_H

// Computes the optimal lambda_max when the input polynomial is a monomial and a maximum number of nodes is required
float128 computeLambdaMax(float128& lambda_min, int num_nodes);

// Computes the number of minimum quadrature nodes by finding the real root of the 7-th degree polynomial equation in (62)
int computeNumNodes(const float128& lambda_min, const float128& lambda_max);

// Computes the order (r) of the monomial map as a linear interpolation of r_min and r_max
double computeMapOrder(const std::vector<float128>& lambdas, const std::vector<float128>& betas);

// Computes the new nodes and weights of the monomial transformation quadrature rule
std::tuple<std::vector<float128>, std::vector<float128>, std::vector<float128>, std::vector<float128>> computeQuadParams(const double& r, const int& n_min, const int& it);

// Computes the numerical integral for a given quadrature rule
template<typename type1, typename type2>
float128 computeQuadRule(const std::vector<type1>& nodes, const std::vector<type1>& weights, std::vector<type2>& muntz_sequence, std::vector<type2>& coeff_sequence);

// Computes the a-posteriori relative error of the quadrature rule
template<typename type>
float128 computeExactError(const float128& In, std::vector<type>& muntz_sequence, std::vector<type>& coeff_sequence, bool& print_primitive);

#endif // MON_TRF_H
```

- `data_management.cpp` is the source file defining each method that does not perform raw computations but instead manages the data flow e.g. in I/O operations. Every function follows the naming scheme **<NameOfFunction>Data** emphasizing its characteristics of data manipulation method. The methods are declared in the header below;

data_management.h

```
#ifndef DATA_MGT_H
#define DATA_MGT_H

// Takes user-defined inputs from file
template<typename type>
std::tuple<int, std::vector<float128>> manageData(std::vector<type>& muntz_sequence, std::
    ↪ vector<type>& coeff_sequence);

// Extract the values of beta_min and beta_max according to the computed minimum number of
    ↪ nodes
std::tuple<int, std::vector<float128>, int> streamMonMapData(const int& comp_num_nodes);

// Degrade the precision of the new nodes and weights to establish minimum data-type for
    ↪ double precision quadrature
template<typename type>
void optimiseData(std::tuple<std::vector<float128>, std::vector<float128>, std::vector<
    ↪ float128>, std::vector<float128>>& quad_params, std::vector<type>& muntz_sequence,
    ↪ std::vector<type>& coeff_sequence);

// Computes and exports the transformed weights and nodes along with other ouputs
template<typename type>
void exportNewData(const std::vector<type>& nodes, const std::vector<type>& weights, const
    ↪ std::vector<float128>& output_data);

#endif // DATA_MGT_H
```

- vector_operations.cpp defines every function that neither performs quadrature-related computations nor I/O data operations. Given their generic nature, no naming scheme is assigned to them as they are declared in homonym header.

vector_operations.h

```
#ifndef VEC_OPS_H
#define VEC_OPS_H

// Returns the float128 input vector in a type specified by the instantiation
template<typename type>
std::vector<type> castVector(const std::vector<float128>& input_vector, const type&
    ↪ type_infer);

// Computes the inner product between two vectors (of the same type) avoiding numerical
    ↪ cancellation
template<typename type>
float128 doubleDotProduct(const std::vector<float128>& f_values, const std::vector<type>&
    ↪ weights);

#endif // VEC_OPS_H
```

The structure of the source code of the library, as organised in the 3 source files presented above, is reported in Figure 2.1. The header files mentioned above are located in the **MTQR/include** subdirectory alongside the primary module's header mtqr.h. Furthermore the library features 3 additional header files, located in **MTQR/data** which contain **tabulated raw data** for the proper construction of the monomial transformation quadrature rule and and thus embedded in the source code at compile time by the C++ preprocessor.

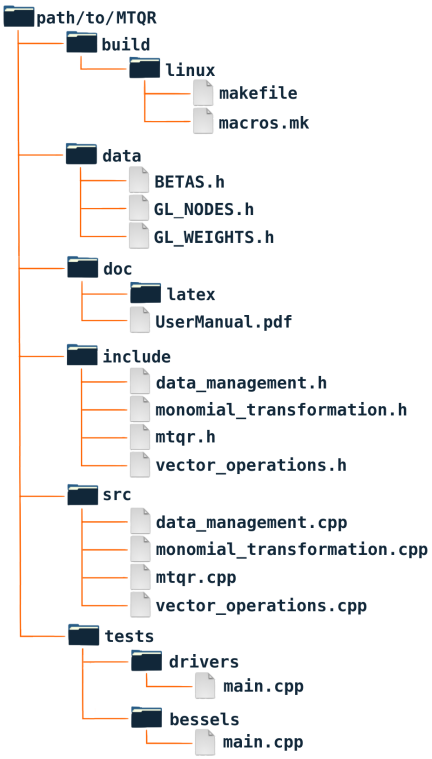

Figure 2.1: The directory tree representing the structure and organisation of MTQR

Those header files are:

- BETAS.h collecting the values of $\beta_{\min}(n)$ and $\beta_{\max}(n)$ for each even value of $n \in [10, 100]$;
- GL_NODES.h storing the G-L quadrature nodes in $(0, 1)$ for each even value of $n \in [10, 100]$;
- GL_WEIGHTS.h storing the weights of the G-L quadrature formula for each even value of $n \in [10, 100]$.

The last two header files store the tabulated values of the original G-L nodes and weights both mapped in $(0, 1)$ with 50 decimal digits of precision (only even-valued number of samples have been stored for the sake of simplicity). Their automatic computation is outside the scope of the library and already explored in details and implemented in other works [5, 9]. The folder **MTQR/build** is used to collect the output files of the build setup instantiated by CMake and the sub-folder **MTQR/build/linux** contains the aforementioned makefile that Linux users can exploit as an alternative to CMake in order to build MTQR.

2.3 BUILD PROCESS

MTQR is a quadrature tool that can be easily integrated in any user applications. As is the case for most C++ open-source software, once the (static) library is compiled succesfully, the user can choose to either create stand-alone applications (as exemplified by the scripts contained in the **MTQR/tests** sub-directory) or use the primary module within a much larger codebase s.a. finite elements solvers and other implemented numerical methods. Regardless of the preferred choice, in order to use MTQR's features the user must not forget to link the desired application against the static library (following the build).

2.3.1 LINUX

In Linux the user can build MTQR by running the autoinstall command. The additional flag all can be appended by the user to also compile the testing applications in **MTQR/tests** (see Section 2.4) as shown below.

Build using the shell script

```
user@machine: MTQR > ./autoinstall all
 *** INSTALLING MTQR ***

g++ -c ../../src/mtqr.cpp -o obj/mtqr.o -g -ansi -std=c++17  -I../../include  -I../../data
g++ -c ../../src/data_management.cpp -o obj/data_management.o -g -ansi -std=c++17  -I../../
    ↪ include  -I../../data
g++ -c ../../src/monomial_transformation.cpp -o obj/monomial_transformation.o -g -ansi -std
    ↪ =c++17  -I../../include  -I../../data
g++ -c ../../src/vector_operations.cpp -o obj/vector_operations.o -g -ansi -std=c++17  -I
    ↪ ../../include  -I../../data
ar rcs ../libmtqr.a obj/mtqr.o obj/data_management.o obj/monomial_transformation.o obj/
    ↪ vector_operations.o
Compiling tests...
...done!
Cleaning object files...
...done!

 *** INSTALLATION COMPLETED ***

 user@machine: MTQR >
```

Alternatively, the user can run `cmake` from the library's root directory and, once completed, execute `make ./build` to create the static library and also compile and link all of the aforementioned testing applications into executables.

Build using CMake

```
user@machine: MTQR> cmake .
-- The CXX compiler identification is GNU 11.3.0
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Found GSL: /usr/include (found version "2.7.1")
-- Found Boost: /usr/lib/x86_64-linux-gnu/cmake/Boost-1.74.0/BoostConfig.cmake (found
    ↪ version "1.74.0")
-- Configuring done
-- Generating done
-- Build files have been written to: /home/MTQR

user@machine: MTQR> make
[  5%] Building CXX object CMakeFiles/mtqr.dir/src/mtqr.cpp.o
[ 11%] Building CXX object CMakeFiles/mtqr.dir/src/data_management.cpp.o
[ 17%] Building CXX object CMakeFiles/mtqr.dir/src/monomial_transformation.cpp.o
[ 23%] Building CXX object CMakeFiles/mtqr.dir/src/vector_operations.cpp.o
[ 29%] Linking CXX static library build/libmtqr.a
[ 29%] Built target mtqr
[ 35%] Building CXX object CMakeFiles/drivers.dir/tests/drivers/main.cpp.o
[ 41%] Linking CXX executable build/drivers
[ 41%] Built target drivers
[ 47%] Building CXX object CMakeFiles/monomial.dir/tests/monomial/main.cpp.o
[ 52%] Linking CXX executable build/monomial
[ 52%] Built target monomial
[ 94%] Building CXX object CMakeFiles/bessel.dir/tests/bessel/main.cpp.o
[100%] Linking CXX executable build/bessel
[100%] Built target bessel

user@machine: MTQR>
```

Regardless of the procedure selected by the user, the static library file libmtqr.a will always be placed in **MTQR/build** therefore, shall the user desire to link custom applications using MTQR such file must be either located in a directory that is included in the PATH variable (e.g. **usr/lib** or **usr/share**) or the full path to libmtqr.a must be specified to the linker.

2.3.2 WINDOWS

We recommend using Microsoft Visual Studio as it automatically detects and runs CMake with the appropriate flags whenever a directory containing a CMakeLists.txt is selected. Below is an example of the output from the terminal window when opening the library's root directory in Visual Studio.

Build using Visual Studio

```
1> CMake generation started for configuration: 'x64-Debug'.
1> Found and using vcpkg toolchain file (C:/vcpkg/scripts/buildsystems/vcpkg.cmake).
1> Command line: "C:\WINDOWS\system32\cmd.exe" /c  ...
1> Working directory: C:\MTQR\out\build\x64-Debug
1> [CMake] -- The CXX compiler identification is MSVC 19.34.31937.0
1> [CMake] -- Detecting CXX compiler ABI info
1> [CMake] -- Detecting CXX compiler ABI info - done
1> [CMake] -- Check for working CXX compiler: C:/Program Files/Microsoft Visual Studio
    ↪ /2022/Community/VC/Tools/MSVC/14.34.31933/bin/Hostx64/x64/cl.exe - skipped
1> [CMake] -- Detecting CXX compile features
1> [CMake] -- Detecting CXX compile features - done
1> [CMake] -- Found GSL: C:/vcpkg/installed/x64-windows/include (found version "2.7.1")
1> [CMake] -- Found Boost: C:/boost (found version "1.81.0")
1> [CMake] -- Configuring done
1> [CMake] -- Generating done
1> [CMake] -- Build files have been written to: C:/MTQR/out/build/x64-Debug/build
1> Extracted CMake variables.
1> Extracted source files and headers.
1> Extracted code model.
1> Extracted toolchain configurations.
1> Extracted includes paths.
1> CMake generation finished.
```

Once CMake terminates the procedure, the user can head to the Solution explorer (most often located on the left panel of Visual Studio), click on the Switch between solutions and available views button and double-click on CMake Targets View. Among the list of all possible targets the user shall first compile the one named mtqr (static library) by left-clicking on it and select Build which, if successful, yields to the following output from the terminal.

Compiling the static library target

```
>------ Build started: Project: CMakeLists, Configuration: Debug ------
  [1/5] Building CXX object CMakeFiles\mtqr.dir\src\mtqr.cpp.obj
C:\boost\boost\multiprecision\cpp_bin_float.hpp(2150): warning C4193: #pragma warning(pop):
    ↪  no matching '#pragma warning(push)'
  [2/5] Building CXX object CMakeFiles\mtqr.dir\src\vector_operations.cpp.obj
C:\boost\boost\multiprecision\cpp_bin_float.hpp(2150): warning C4193: #pragma warning(pop):
    ↪  no matching '#pragma warning(push)'
  [3/5] Building CXX object CMakeFiles\mtqr.dir\src\data_management.cpp.obj
C:\boost\boost\multiprecision\cpp_bin_float.hpp(2150): warning C4193: #pragma warning(pop):
    ↪  no matching '#pragma warning(push)'
  [4/5] Building CXX object CMakeFiles\mtqr.dir\src\monomial_transformation.cpp.obj
C:\boost\boost\multiprecision\cpp_bin_float.hpp(2150): warning C4193: #pragma warning(pop):
    ↪  no matching '#pragma warning(push)'
C:\MTQR\src\monomial_transformation.cpp(105): warning C4244: '=': conversion from 'double'
    ↪ to 'int', possible loss of data
  [5/5] Linking CXX static library mtqr.lib
```

Following the above the mtqr.lib file will be available in **MTQR\out\build\x64-Debug\build** sub-directory. At this point the user is ready to link any application to the static library or running the proposed test cases shipped with the library's source code (see the following Section).

2.4 TESTING THE BUILD

After the successful compilation of the library the user is ready to instantiate MTQR's primary module in any desired application. It is however a widespread good-practice to test the build of an installed library before proceeding with its usage and this principles becomes particularly truthful in the case of MTQR. For that purpose we prepared few test applications, located in the aforementioned **MTQR/tests** which not only satisfy the sanity-check principle for the build but, in our opinion, also provide the user with an aid on how to use MTQR via its primary module and therefore ease its implementation in larger codebase. We remark that these tests have been successfully ran on all the platforms listed in Table 1.2; given the compatibility with multiple architectures we will only refer to the results obtained and omit the specifications for the system on which they have been derived since they do not vary. Furthermore we reiterate that if the library has been built using CMake the executables of the test cases will be located in **MTQR/build** on Linux systems (**MTQR/build/linux** if compiled using autoinstall all) and in **MTQR\out\build\x64-Debug\build** on Windows (using Microsoft Visual Studio). For the sake of completeness we show how the drivers test is built on Windows while we compile and execute the bessels test on Linux.

2.4.1 DRIVERS

In the **MTQR/tests/drivers** sub-directory the main.cpp file contains a very straight-forward yet powerful example of how an application can be built by the user with MTQR. It presents three tests cases where we want to integrate three functions that are very different among each others. The three polynomial functions are the following

$$
\begin{aligned}
p_1(x) &= 5x^{-\frac{\pi}{4}} - x^{-\frac{1}{2}} + 1 + 10x^2 + ex^{e+\frac{1}{4}} \\
p_2(x) &= x^{-\frac{e}{3}} \\
p_3(x) &= x^{17} + x^{35}
\end{aligned}
$$

and, with the exception of $p_3(x)$, all have a singularity in $x = 0$ (MTQR defaults the integration interval to be $(0, 1)$). Using Visual Studio the workflow would be first to compile the executable associated to the present test case, which is accomplished by repeating the exact same steps outlined in Sub-Section 2.3.2 except now they are done for the target drivers (executable) listed in the Solution explorer alongside the other executable targets.

Compiling the drivers test case

```
[1/2] Building CXX object CMakeFiles\drivers.dir\tests\drivers\main.cpp.obj
C:\boost\boost\multiprecision\cpp_bin_float.hpp(2150): warning C4193: #pragma warning(pop):
    ↪  no matching '#pragma warning(push)'
[2/2] Linking CXX executable drivers.exe
```

Once compiled, the executable can be run within Visual Studio itself by simply opening the integrated PowerShell terminal, navigating to **MTQR\out\build\x64-Debug** and running the drivers.exe file. As evidenced below, the resulting application integrates each polynomial using the monomial transformation quadrature rule provided by MTQR through the primary module. We remark that, although no difference in numerical results occurs, the following report solely concerns the loud mode of execution. A list of information regarding the monomial transformation itself is displayed on the terminal whereas the results of interest (i.e. the new processed quadrature together with the related numerical integration) are exported in files located in the appropriate output directory (see Section 3.2).

Executing the driver test case: p_1(x)

```
PS C:\MTQR> cd .\out\build\x64-Debug\
PS C:\MTQR\out\build\x64-Debug> .\drivers.exe

    |***********************************************|
    |                  ** MTQR **                   |
    |  ** MONOMIAL TRANSFORMATION QUADRATURE RULE **  |
    |***********************************************|


 Input polynomial p(x) =  +2.71828183*x^(2.96828183) +5*x^(-0.785398163) -1*x^(-0.5) +x^(0)
    ↪   +10*x^(2)

 ** Accepted sequence of exponents **
    {2.96828183, -0.785398163, -0.5, 0, 2}
 ** Lambda_min = -0.785398163, Lambda_max = 2.96828183 **
 ************************************************
 ** N_min = 32
 ** Beta_min = 4.37782519, Beta_max = 127.894326 **
 ** Transformation order = 28.7703455 **
 ************************************************
 ** Using double f.p. format for nodes and weights **
 ** I(p(x))   = 26.317297376488324 **
 ** I_n(p(x)) = 26.317297376488324  [with parameters in float128 precision] **
 ** E_n(p(x)) = 1.3889468142066847e-17  [with parameters in float128 precision] **
 ** I_n(p(x)) = 26.317297376488324  [with parameters in double precision] **
 ** E_n(p(x)) = 5.4523153678583708e-18 [with parameters in double precision] **
```

The results of the quadrature and the new nodes and weights are then exported in three separate files called Results.txt, Nodes.txt and Weights.txt all located in a **output** directory created automatically at runtime in the same path of the user's executable. To run the second and third benchmark the user needs to press ENTER on whatever terminal the process is being paused on. Note that, since we are implementing three benchmark polynomials in a single source file, such pause is necessary during the execution in order to capture the new processed quadrature together with the associated numerical integration as the files in the output directory are overwritten at each execution. The correct execution of all three benchmarks will then close the application. The user should check that each benchmark is executed without problems and the results are correctly generated (see Section 3.2). As for the quality of the results obtained for those tests, an in-depth analysis, coupled with the fast execution of the program, shows the advantages provided by MTQR over classical G-L and other generalised quadrature rules. The first polynomial $p_1(x)$ is a model proposed in equation (73) of [11], *Section 5.1, Example 1*, and it represents a a typical integrand function in numerical methods for differential and integral equations featuring singular modelling, on which MTQR is indeed considered an helpful, effective and precise tool for integration. The proposed generalised polynomial $p_1(x)$ features a singularity at $x = 0$ in the interval $(0, 1)$ and all but two of its monomial terms have non-integer degree. The numerical approximation of the integral at double precision is achieved with $n = 32$ nodes using a monomial transformation applied to the classical G-L quadrature with n nodes in $(0, 1)$. The Table 2.2 below shows what was graphically reported in Figure 1.2 i.e. the ad-hoc suited and more efficient distribution of the samples obtained by the monomial transformation as opposed to the ones of the classical G-L quadrature rule. While in fact we know that the $n = 32$ nodes of the classical G-L quadrature rule are distributed symmetrically in $(0, 1)$ with an equal abundance of those at both bounds of the integration interval, our proposed quadrature scheme, based on the monomial transformation, concentrates the samples close to the lower bound $x = 0$, squashing the vast majority of them (26 out of the total 32) inside the sub-interval $(0, 0.1)$ thereby better capturing the singular behaviour of $p_1(x)$. Notice how, while instantiating the primary module for $p_1(x)$, we generate a precision quadrature for all generalised polynomials with sets of $\lambda \in [\lambda_{min}, \lambda_{max}] = [-\pi/4, e + 1/4]$. The second benchmark polynomial is proposed with the scope of introducing the user to a core functionality of MTQR that requires further input on the CLI. Indeed, $p_2(x)$ is a monomial of non-integer degree and its numerical integration requires careful manipulation. During the modelling process of larger applications, especially when dealing with finite elements constructed over meshed domains with singular spatial geometry behaviour [7, 8] it is often necessary to integrate low-order singular generalised polynomial basis functions together with high-order regular polynomial basis functions.

Nodes and weights of the proposed quadrature for $p_1(x)$		
$j \in \mathbb{N}$	$x_j \in (0,1)$	$w_j \in \mathbb{R}^+$
1	4.0256721894941735e-83	2.9709584266857193e-81
2	2.2116841854653406e-62	7.1971053989801097e-61
3	3.4370358897566318e-51	7.1255611223692978e-50
4	1.6010758830624544e-43	2.4253789034229506e-42
5	1.0479208102676717e-37	1.2456363438765983e-36
6	4.8718071131884711e-33	4.7453741025535338e-32
7	3.7119919463646725e-29	3.0494845207591382e-28
8	7.5510919371932376e-26	5.3393428605147779e-25
9	5.58947068987428e-23	3.4526789629984772e-22
10	1.8549840400142102e-20	1.0121884313459638e-19
11	3.1981531227726135e-18	1.5545919457374846e-17
12	3.1905126409913618e-16	1.3904418738462426e-15
13	1.9974633805093215e-14	7.8421013950383668e-14
14	8.3548669580378659e-13	2.9653252968844048e-12
15	2.4525558094710645e-11	7.8879379262599627e-11
16	5.2553535644885825e-10	1.5337625816699302e-09
17	8.4865615532959245e-09	2.2485149221054993e-08
18	1.0601166175497224e-07	2.5487504629916981e-07
19	1.0467771915132323e-06	2.2805293469267595e-06
20	8.3187859060800558e-06	1.6383778757563812e-05
21	5.4017656350748472e-05	9.583886949817516e-05
22	0.00029027719974030597	0.00046172220499588948
23	0.0013048618829373392	0.0018488842315729846
24	0.0049515587664587246	0.0061972140073356637
25	0.01593869201550067	0.017474549464975012
26	0.044176504650425052	0.041563995305652503
27	0.10510275956828145	0.083385319613667491
28	0.21621415218381346	0.14051580648196024
29	0.38598199665949656	0.19670495174814193
30	0.59964365548997856	0.22295968412650397
31	0.81242716007987004	0.1915755472071223
32	0.96137880664253184	0.097197543454586643

Table 2.2: Nodes and weights obtained by a monomial transformation of order $r = 28.77$ applied to the G-L quadrature integrating $p_1(x)$ in $(0,1)$ at double precision with $n = 32$ samples.

MTQR allows to integrate such integrals identifying the degree of the monomial as $\lambda_{\min}$ and the maximum degree $\lambda_{\max}$ of the all set of combined polynomial functions. For this reason, in this test, we add a monomial with $\lambda_{\max}$ degree and unitary coefficient to $p_2(x)$. Once constructed the final polynomial we apply the proposed algorithm to process the optimised nodes and weights. The integration is enabled using a caveat prompted on the terminal and it consists in choosing one path between two mutually exclusive options that the user must exercise in order to continue with the application.

Executing the driver test case: p_2(x)

```
   |***********************************************|
   |                ** MTQR **                     |
   |  ** MONOMIAL TRANSFORMATION QUADRATURE RULE **  |
   |***********************************************|

 Input polynomial p(x) =  +x^(-0.906093943)

** WARNING ** Your input is a monomial of non-integer degree.
              MTQR needs a binomial for double-precision quadrature.
              How do you proceed? ['nodes' for n_min ~ 'lambda' for lambda_max]
              Input:
```

One of those is the choice of specifying the maximum number n for the quadrature rule, in which case the library automatically compute $\lambda_{\max}$ and integrate $\tilde{p_2}(x) = x^{-\frac{6}{3}} + x^{\lambda_{\max}}$ accordingly. The second path allows instead to specify $\lambda_{\max}$ directly and let MTQR carry out the integration. It is easy to see that the second case reduces to a *"standard"* polynomial input, as far as the library is concerned. The user should therefore type nodes on the CLI and press ENTER; we are now prompted to select any value of $n \in [10, 100]$; user should type, for example, 64 and press ENTER again. On the contrary typing lambda causes the application to resort to its original workflow with the additional intermediate step of requiring the input value of $\lambda_{\max}$. We remark that any miss-typed or empty input results in MTQR to throw an error message and exit the program.

Building and executing the test driver: p_2(x)

```
Please specify the desired number of quadrature nodes (number must be even): 64
 ************************************************
 ** N_min = 64
 ** Beta_min = 3.25021668, Beta_max = 511.19448 **
 ** Accepted sequence of exponents **
    {-0.906093943, 10.3166381}
 ** Lambda_min = -0.906093943, Lambda_max = 10.3166381 **
 ************************************************
 ** N_min = 64
 ** Beta_min = 3.25021668, Beta_max = 511.19448 **
 ** Transformation order = 45.2603038 **
 ************************************************
 ** Using double f.p. format for nodes and weights **
 ** I(p(x))   = 10.737305800857456 **
 ** I_n(p(x)) = 10.737305800857441     [with parameters in float128 precision] **
 ** E_n(p(x)) = 1.424785764850455e-15  [with parameters in float128 precision] **
 ** I_n(p(x)) = 10.737305800857441     [with parameters in double precision] **
 ** E_n(p(x)) = 1.4344708669352776e-15 [with parameters in double precision] **

 ** MTQR HAS TERMINATED **
```

After the specified selection, MTQR computes automatically a value of $\lambda_{\max} = 10.3166381$ for $p_2(x)$ and the resulting quadrature barely retains the machine-epsilon double precision for the relative error of the numerical integration. Here, the library is stressed more as it can be evinced by the substantially greater value of the transformation order ($r = 45.2603038$) w.r.t. the previous instance in $p_1(x)$. The effects of such magnitude for the monomial transformation order are easily registered by assessing the mapped nodes and weights listed in the following table. Furthermore we highlight how the relative error obtained with float128 monomial transformation quadrature rule's parameters is not substantially better than the one derive with the same nodes and weights optimised in double f.p. format. In fact we observe that the latter format has been selected despite being slightly less precise than the former. The logic behind is that both relative errors are computed at run-time and if their absolute difference is smaller than twice the machine-epsilon in double precision than we go ahead and export the data in the most optimised format. Finally, with the last benchmark a classical polynomial of high integer-degree is integrated; indeed, $p_3(x)$ is a binomial. For standard, polynomials we compare the performance of standard G-L quadrature with the one of MTQR.

Nodes and weights of the proposed quadrature for $p_2(x)$					
$j \in \mathbb{N}$	$x_j \in (0,1)$	$w_j \in \mathbb{R}^+$	$j \in \mathbb{N}$	$x_j \in (0,1)$	$w_j \in \mathbb{R}^+$
1	2.7630147554775799e-157	3.2089404684705014e-155	33	7.0500634584281311e-14	1.5167370865604578e-13
2	1.2504818169246583e-124	6.4130575288015817e-123	34	5.7559583188242479e-13	1.1793837946815465e-12
3	5.6814585252048909e-107	1.8611939310882479e-105	35	4.2517694749483687e-12	8.2956748948158666e-12
4	7.8015395380491066 8e-95	1.8746032945680622e-93	36	2.8538360103496875e-11	5.3006402608938980e-11
5	1.3770579298628561 6e-85	2.6096113660080927e-84	37	1.7474698043212560e-10	3.0884893242340361e-10
6	3.9955200879667989e-78	6.2503681667652076e-77	38	9.7966474232194539e-10	1.6467041924026613e-09
7	7.1241214820356870 4e-72	9.4622687681839969e-71	39	5.0450425037572576e-09	8.0595541550951593e-09
8	1.6775862111624234 0e-66	1.9352874634013771e-65	40	2.3937780380919019e-08	3.6314704984080236e-08
9	8.5039480424641786 2e-62	8.6610446966327004e-61	41	1.0493924006829552e-07	1.5103117353143750e-07
10	1.2893742043052028 2e-57	1.17417554134314e-56	42	4.2611947777521824e-07	5.8115902014625234e-07
11	7.3769554331428493 0e-54	6.0678528535818764e-53	43	1.6064924620227659e-06	2.0734907170162103e-06
12	1.8882385421558174 4e-50	1.4144870181471483e-49	44	5.6352337753679891e-06	6.8727595063813297e-06
13	2.4587404179122132 3e-47	1.6888846224093984e-46	45	1.8428406102394049e-05	2.1200152242057246e-05
14	1.7987909549913160 1e-44	1.1394463566176748e-43	46	5.6285102084355121e-05	6.0953635295967037e-05
15	7.9964475542787055 9e-42	4.6939290576395100e-41	47	0.000160824501490444	0.00016357104996692
16	2.3003183455086942 5e-39	1.256455692921632e-38	48	0.000430554366218858	0.000410178545783102
17	4.5073622443142326 6e-37	2.2990244539015347e-36	49	0.001081507980701627	0.00096212522184789
18	6.2760501764191665 3e-35	2.9984883056622040e-34	50	0.002552202028188329	0.002112665166000562
19	6.4332966119326743 3e-33	2.8866958399889620e-32	51	0.005664918590851289	0.004345383033450631
20	5.0016798664394982 0e-31	2.1127162036193374e-30	52	0.011839418057758923	0.008374932123081819
21	3.0252254469427907 3e-29	1.2053618210763572e-28	53	0.023321020012667499	0.015162268661501289
22	1.4548513213928400 0e-27	5.4774213217611214e-27	54	0.043333892814950261	0.025596514741963807
23	5.6685777978474631 5e-26	2.0197293557945103e-25	55	0.076017823001042698	0.040555711474099192
24	1.8190161317795624 4e-24	6.1417478949789698e-24	56	0.1259851142914331	0.060094184332119245
25	4.8767441078562962 2e-23	1.5621281667244454e-22	57	0.197384568387222875	0.083114399815460055
26	1.1061918811681505 6e-21	3.3648881412038240e-21	58	0.292508401834271963	0.106964653637029898
27	2.1467416970420781 7e-20	6.2062442929374053e-20	59	0.410207778274644215	0.127463313674079491
28	3.5997972451025567 9e-19	9.8975903331295861e-19	60	0.544613347654060773	0.139504287218888873
29	5.2621101787069895 5e-18	1.3767307083491291e-17	61	0.68476273369408191	0.138243738177407818
30	6.7586203469217497 2e-17	1.68331269687316e-16	62	0.815603424165963892	0.120596149243306058
31	7.6816681240557070 0e-16	1.8218265936883635e-15	63	0.920443566774952144	0.086540144565319482
32	7.7755377170485257 9e-15	1.756314510825820e-14	64	0.984393322175774548	0.039739900698273792

Table 2.3: Nodes and weights obtained by a monomial transformation of order $r = 45.26$ applied to the G-L quadrature integrating $p_2(x)$ in (0,1) at double precision with $n = 64$ samples.

According to the properties described in Sub-Section 1.2.2, the former requires $n = \frac{\lambda_{max}+1}{2} = 18$ nodes to achieve double precise integration however, we immediately realise that our library outperforms the classical G-L quadrature rule for a fixed double precision accuracy as only $n = 12$ samples are needed. We also remark that the transformation order needed by the monomial map is significantly lower than the previous two cases, in particular less than 1. Since $p_3(x)$ does not have a singularity at $x = 0$, the processing of G-L quadrature yields a monomial transformation quadrature with samples that are not clustered on $x = 0$. In this instance instead, the transformation forces the performance of the quadrature to effectively integrate all generalised polynomials with $\lambda \in \{17, 35\}$ at machine-epsilon precision specified for the double f.p. format, as it can be assessed by the table below. With these results we would like to direct the user's attention on the possibility of using MTQR even for specific standard/classical polynomials of integer degree as its processed nodes and weights lead to a fewer samples in the quadrature rule (especially with higher-degrees) and thus more efficient and less expensive code to run. At the best of our knowledge, and based on the tests reported here and in [11, 12], the monomial transformation can be adapted to the widest range of generalised polynomial of non-integer degree as long as the constraint of $\lambda_{\min} > -1$ holds for the integrand.

```
Executing the driver test case:  p_3(x)

    |***********************************************|
    |                 ** MTQR **                    |
    |  ** MONOMIAL TRANSFORMATION QUADRATURE RULE **  |
    |***********************************************|

Input polynomial p(x) =  +x^(17)  +x^(35)

 ** Accepted sequence of exponents **
    {17, 35}
 ** Lambda_min = 17, Lambda_max = 35 **
 ************************************************
 ** N_min = 12
 ** Beta_min = 8.54130275, Beta_max = 23.2002133 **
 ** Transformation order = 0.601150262 **
 ************************************************
 ** Using double f.p. format for nodes and weights **
 ** I(p(x))   = 0.083333333333333333 **
 ** I_n(p(x)) = 0.083333333333333326    [with parameters in float128 precision] **
 ** E_n(p(x)) = 8.8727918916220005e-17  [with parameters in float128 precision] **
 ** I_n(p(x)) = 0.083333333333333323    [with parameters in double precision] **
 ** E_n(p(x)) = 1.1803389750004119e-16  [with parameters in double precision] **

 ** MTQR HAS TERMINATED **

PS C:\MTQR\out\build\x64-Debug>
```

By excluding those cases of rational functions we argue that the usage of MTQR produces more accurate results faster (i.e. using the minimum possible number of samples) and more efficiently (outputting the most optimised f.p. formats for the new quadrature parameters) than any other algorithm that currently deals with both classical and singular generalised polynomial integrands.

Nodes and weights of the proposed quadrature for $p_3(x)$		
$j \in \mathbb{N}$	$x_j \in (0,1)$	$w_j \in \mathbb{R}^+$
1	0.0597707229696358	0.0919264663553836
2	0.1610307309568925	0.1079664407846363
3	0.2725515941208583	0.1139863183561349
4	0.3872246099856674	0.1145999654489373
5	0.5003873977306973	0.1111039615531504
6	0.6082809536783961	0.1041479683059256
7	0.7076854327640707	0.0941968566214454
8	0.7958143666363263	0.0816648573999376
9	0.8702918448156044	0.0669634924046180
10	0.9291601555462656	0.0505192348221430
11	0.9708981507831240	0.0327793590987762
12	0.9944473508291223	0.0142322139984140

Table 2.4: Nodes and weights obtained by a monomial transformation of order $r = 0.6011$ applied to the G-L quadrature integrating $p_3(x)$ in (0,1) at double precision with $n = 12$ samples.

2.4.2 BESSEL

We now address the building process for applications using MTQR on Linux by compiling and executing the **MTQR/tests/bessels** numerical example that integrates Bessel functions of fractional order [1] which are expressedn with a series expansion of the form

$$J_\nu(x) = \left(\frac{1}{2}x\right)^\nu \sum_{m=0}^{+\infty} \left(\frac{(-\frac{1}{4}x^2)^m}{m!\,\Gamma(\nu+m+1)}\right), \quad \nu \in \mathbb{Q}$$

where $\Gamma(z) = \int_0^{+\infty} x^{z-1}e^{-x}dx$ is Euler's gamma function. As with the previous test, the standalone main.cpp script treats three separate integrands, specifically

$$b_1(x) = J_{-\frac{1}{2}}(x)\,, \quad \left[\lambda_{\min} = -\frac{1}{2},\, \lambda_{\max} = \frac{15}{2}\right]$$
$$b_2(x) = J_{-\frac{1}{2}}(x) + J_{\frac{1}{3}}(x)\,, \quad \left[\lambda_{\min} = -\frac{1}{2},\, \lambda_{\max} = \frac{25}{3}\right]$$
$$b_3(x) = J_{-\frac{1}{2}}(x)\, J_1(x)\,, \quad \left[\lambda_{\min} = \frac{1}{2},\, \lambda_{\max} = \frac{33}{2}\right]$$

all truncated up to the first $m = 5$ terms. As explained in Sub-Section 2.3.1 compiling the library using the built-in command `autoinstall all` will make the executable of both tests readily available. Therefore, from the **MTQR/build/linux** sub-directory we can easily launch the `bessels` executable

Executing the bessels test case: b_1(x)

```
user@machine: MTQR/build/linux > ./bessels
Iteration n. 1:  In=1.4434118505832414, En=1.3842419943602952e-09   (n. samples=18)
Iteration n. 2:  In=1.4434118485851772, En=2.2767307585229809e-14   (n. samples=24)
Iteration n. 3:  In=1.4434118485852097, En=3.0766631871932174e-16   (n. samples=28)
Iteration n. 4:  In=1.4434118485852099, En=1.5383315935966087e-16   (n. samples=32)
```

In this (and also the other examples of the `bessels` tests) we perform 4 iterations in which the integrand is truncated with increasingly more terms in the series expansion; so for the first iteration we only consider the first $m = 2$ terms of the series, at the second iteration we consider a slightly more accurate truncation with $m = 3$ terms and so on until we reach the $m = 5$ terms truncation at the fourth and last iteraton. As printed on the terminal, it is evident that, while using MTQR, only $m = 4$ terms are necessary to achieve double machine precision in the integration of $b_1(x)$ using $n = 28$ samples. Notice that in this test we are instantiating MTQR's silent mode which, as addressed later in Section 3.3, exports the optimised quadrature samples at runtime while also parsing them (in double f.p. format) in a single `outputs.txt`. Such nodes and weights are reported in Table 2.5 for the specific case of $b_1(x)$. Pressing `ENTER` on the terminal will cause the program to progress to the integration of $b_2(x)$.

Executing the bessels test case: b_2(x)

```
Iteration n. 1:  In=2.0616765364427128, En= 1.5093001397369456e-11   (n. samples=20)
Iteration n. 2:  In=2.0616765364115950, En= 4.3080396173398953e-16   (n. samples=26)
Iteration n. 3:  In=2.0616765364115954, En= 2.1540198086699477e-16   (n. samples=30)
Iteration n. 4:  In=2.0616765364115959, En= 0   (n. samples=32)
```

whose results are reported in Table 2.6; in this case we achieve double precision already at the second iteration with $m = 3$ terms approximated using $n = 26$ samples in the quadrature formula. Pressing `ENTER` once more the final generalised polynomial $b_3(x)$ will be integrated

Executing the bessels test case: b_3(x)

```
Iteration n. 1:  In=0.20217936665914402, En= 4.6538577547142047e-14   (n. samples=16)
Iteration n. 2:  In=0.20217936665913455, En= 2.7456387933417138e-16   (n. samples=20)
Iteration n. 3:  In=0.20217936665913452, En= 4.1184581900125707e-16   (n. samples=24)
Iteration n. 4:  In=0.20217936665913455, En= 2.7456387933417138e-16   (n. samples=26)
```

which can be checked against Table 2.7. Once again we reach double precision at the second iteration ($m = 3$) with $n = 20$ samples.

Nodes and weights of the proposed quadrature for $b_1(x)$		
$j \in \mathbb{N}$	$x_j \in (0,1)$	$w_j \in \mathbb{R}^+$
1	2.4886199426064151e-37	8.4962373994555118e-36
2	9.7331836637461709e-28	1.4643479611244592e-26
3	1.4457358100950039e-22	1.3842522302572618e-21
4	5.0016005668192019e-19	3.4937303742677519e-18
5	2.3898895696221016e-16	1.3072205683995924e-15
6	3.3379176999637574e-14	1.4921737796227685e-13
7	2.0061814162880028e-12	7.5396111250023958e-12
8	6.5024461302093438e-11	2.0952062117980374e-10
9	1.3114937396033565e-09	3.6747077855002399e-09
10	1.8127051625051056e-08	4.4623882566268086e-08
11	1.8378134371548134e-07	4.0048601618886300e-07
12	1.4364127550661674e-06	2.7856943512053638e-06
13	8.9845181012671335e-06	1.5562277176261812e-05
14	4.6284863157369641e-05	7.1752397117410841e-05
15	0.00020086526821504015	0.00027887690430810123
16	0.00074763338591703199	0.00092899238516954601
17	0.00242144448149024960	0.00268729403322269480
18	0.00690521329493283800	0.00681984845919765200
19	0.01750612415771758700	0.01530440169550289400
20	0.03977115705124230300	0.03054591754739828500
21	0.08150130998495218300	0.05442760391046814300
22	0.15147217453530576000	0.08671671604631447900
23	0.25645214394084537000	0.12340206986099296000
24	0.39697030548729967000	0.15606137516867771000
25	0.56343880239103239000	0.17333540795894398000
26	0.73494083607061778000	0.16470223026049224000
27	0.88247644045564666000	0.12528804760124707000
28	0.97658034792995352000	0.05941066879760208800

Table 2.5: Nodes and weights used to integrate $b_1(x)$ in $(0,1)$ at double precision with $n = 28$ samples.

Nodes and weights of the proposed quadrature for $b_2(x)$		
$j \in \mathbb{N}$	$x_j \in (0,1)$	$w_j \in \mathbb{R}^+$
1	3.4516356760812973e-38	1.2342525913121972e-36
2	3.8325772723471142e-28	6.0369566088450707e-27
3	9.9225299473239260e-23	9.9396566159073773e-22
4	4.9825453071219598e-19	3.6374027690472840e-18
5	3.1314033756995992e-16	1.7875170228738393e-15
6	5.4014064204524397e-14	2.5153717600790947e-13
7	3.8341790457768706e-12	1.4977141913711563e-11
8	1.4186407199801465e-10	4.7382771570885528e-10
9	3.1782708841876993e-09	9.2009563973473080e-09
10	4.7693544489375830e-08	1.2083866041878865e-07
11	5.1471873835507824e-07	1.1491387231685599e-06
12	4.2080094103272480e-06	8.3155689624382145e-06
13	2.7096647668129325e-05	4.7518363607941460e-05
14	0.00014160947264627827	0.00022056232107464771
15	0.00061486044481872467	0.00084984903258738479
16	0.00225961945233026230	0.00276441941209095270
17	0.00713409812235353110	0.00769039008222996080
18	0.01958482865239455400	0.01847606567892801100
19	0.04720943444309261300	0.03859904705635783300
20	0.10072245182998968000	0.07040611356048015500
21	0.19143513145042113000	0.11222218641959783000
22	0.32582069201667130000	0.15576465398114248000
23	0.49864856900182841000	0.18628105160570962000
24	0.68842920718272826000	0.18714459947457029000
25	0.85940977813209840000	0.14789816858200941000
26	0.97169527987442395000	0.07162577919325288600

Table 2.6: Nodes and weights used to integrate $b_2(x)$ in $(0,1)$ at double precision with $n = 26$ samples.

Nodes and weights of the proposed quadrature for $b_3(x)$		
$j \in \mathbb{N}$	$x_j \in (0,1)$	$w_j \in \mathbb{R}^+$
1	4.1953101534401230e-13	5.4020999989060756e-12
2	1.7277489396940095e-09	9.7806246482524820e-09
3	1.5131802891266276e-07	5.4278275162065674e-07
4	3.1764543565866453e-06	8.2592720442587919e-06
5	3.1283870913219795e-05	6.3145562319811638e-05
6	0.00019178932370735830	0.00031287448049440619
7	0.00084675725874190579	0.00114515937670795580
8	0.00293097461726764260	0.00334042491854184740
9	0.00839316199004415740	0.00814448626153404210
10	0.02061417523469090800	0.01712844924548821000
11	0.04453655482458510100	0.03174449292872995300
12	0.08620534045991272400	0.05261231257702993900
13	0.15153905872038897000	0.07874057992635298500
14	0.24442636156584369000	0.10701773980543285000
15	0.36459334623544015000	0.13231047685467093000
16	0.50595731092684415000	0.14834580910696535000
17	0.65622026419146040000	0.14926214181157157000
18	0.79818281920658385000	0.13140769321947060000
19	0.91273093247328052000	0.09478387966154840100
20	0.98286045029632330000	0.04363152242235273500

Table 2.7: Nodes and weights used to integrate $b_3(x)$ in $(0,1)$ at double precision with $n = 20$ samples.

Chapter 3

USER INTERFACE

In this chapter we briefly discuss the input and output user interface of the library. In the first section we discuss the former aspect by showing the appropriate data structures to be deployed in any custom application instantiating the primary module of MTQR. In the second section we address the outputs of the library and how the results are exported; we also consider the main differences between the loud and silent mode of execution. Subsequently we briefly outline how MTQR's results can be used by other applications in larger software. Finally we conclude by giving an exposition of the methods instantiated within the primary module.

3.1 THE INPUT SOURCE FILE

Users of MTQR can either choose to access its features via the access point provided by the primary module or use some of its other methods without additional changes to the source code and compile options; this is due to the fact that all the methods declarations are included in the library principal header (see Section 2.1). We postpone the former option to Section 3.4 while in the following we focus on a brief description of the correct instantiation of the primary module in the user's source code. In the default loud mode (see Section 3.3 for the silent mode) the primary module `mtqr` is invoked by specifying its two inputs which uniquely identify the generalised polynomial function that the user desires to integrate numerically:

- the `coefficients_sequence` stored in a `std::vector` of length $r+1$, that is the number of non null-coefficients;
- the `muntz_sequence` of exponents also stored in a `std::vector` of the same length $r+1$.

We remark that if the two data-structures have different lengths, MTQR throws an error message and exits the program. Moreover the order of input of either `muntz_sequence` and `coefficients_sequence` does not matter as MTQR automatically extract $\lambda_{\min}$ and $\lambda_{\max}$ through a *sorting* algorithm. A minimalist example of correct instantiation of the primary module in loud mode is given below of sub-section 2.4.1.

Minimal working example (loud mode)

```
#include "mtqr.h"

int main(int argc, char** argv)
{
   std::vector<float128> coeff_sequence = {E, 5.0, -1.0, 1.0, 10.0};
   std::vector<float128> muntz_sequence = {E + 0.25, -PI/4, -0.5, 0, 2};
   mtqr(muntz_sequence, coeff_sequence);

   return 0;
}
```

Although the aim of MTQR is to integrate generalised polynomials with double precision, we note that the coefficients and exponents sequences are defined as float128 data-types; this is to ensure the convergence check on our tests without losing f.p. precision. Applications with coefficients and exponents sequences defined with double precision is straightforward.

3.2 RESULTS AND OUTPUTS

Now we discuss the outputs of MTQR and how the user can retrieve and use them in custom applications. Upon invoking the primary module in loud mode, the first feedback to the user is its input followed by the computed parameters of the monomial map i.e. $\lambda_{\min}$, $\lambda_{\max}$, $\beta_{\min}$, $\beta_{\max}$, r. The next information is the floating-point format with which the new nodes and weights have been exported and lastly the value of the primitive (or analytic integral), numerical integral and the relative error obtained using the new quadrature parameters in quadruple (i.e. float128 f.p. format) and double precision. The classical G-L quadrature parameters are stored as std::string hard data in GL_NODES.h and GL_WEIGHTS.h which are then embedded in the source code as float128 f.p. data-types, retaining up to 34 decimal digits of precision. Since double-precision quadrature can be achieved with lower precision, MTQR features a method called optimiseData that automatically selects the most optimised format possible between, float128 and double, with which to export the new quadrature nodes and weights to assures the prescribed relative precision. The optimality here is meant as the lowest-precision f.p. format that still allows to retain a machine-epsilon accuracy (we specified double f.p. precision however the procedure is easily generalised using C++ template parameters) for the relative error of the integral computed through the monomial transformation quadrature rule. The results of the quadrature and the new nodes and weights are then exported in three separate files called Results.txt, Nodes.txt and Weights.txt, all located in a **output** subdirectory created automatically at runtime in the same path of the user's executable. These text files collect the inputs and outputs of the library alongside the values of the numerical integral approximated using both the classical G-L and the proposed monomial transformation quadrature rule together with the exact analytical result. The file Results.txt is therefore intended for the user to have an immediate feedback on the quality of the approximation made by MTQR. The remaining two files, as their names suggest, list the actual output of the library i.e. the new nodes and weights respectively. The resulting new quadrature samples, established by the monomial transformation quadrature rule and optimised accordingly by the aforementioned routine, are exported with the f.p. precision that guarantees the a-posteriori relative error of the numerical integral to be within the machine-epsilon in double precision. We recall that the quadrature rule built with this program allows precise integration of any generalised polynomials with real-valued degrees $\lambda \in [\lambda_{min}, \lambda_{max}]$. When executing MTQR in loud mode, as we did for the testing applications in Section 2.4, those new quadrature parameters are not streamed to the memory as outputs of the primary method mtqr; they can only be retrieved manually (at any time) by the user by accessing the aforementioned text files. There might be several applications however in which it is not ideal to export the optimised, transformed quadrature samples into text files and re-loading them back to memory to be used within the same source code (e.g. finite element solvers). For this reason, as already mentioned in several occasions throughout the user manual thus far, the primary module mtqr is overloaded with a silent mode of execution, which we refer to as the **service routine**, that allows for much more efficient interface of MTQR's output and the user's application.

3.3 SERVICE ROUTINE

As opposed to the canonical loud execution discussed thus far, the silent mode only takes two double parameters specifying $\lambda_{\min}$ and $\lambda_{\max}$ instead of the full sequence of exponents and coefficients. The silent mode then defaults both generalised monomials to have unitary coefficients and thus computes the quadrature of the following integrand

$$p(x) = x^{\lambda_{\min}} + x^{\lambda_{\max}}, \quad x \in (0,1)$$

using the exact same monomial transformation quadrature rule. As opposed to the loud mode discussed and instantiated thus far, the silent mode overloading MTQR does in fact return the new nodes and weights to the stack/heap of the user's local machine, returned by the function in a std::vector<std::vector<double> > data structure as depicted in the snippet below. Once again we propose a minimalist example for specific input values of $\lambda_{\min} = -\frac{1}{e}$ and $\lambda_{\max} = +\frac{1}{2}$ reported below.

Minimal working example (silent mode)

```
#include "mtqr.h"

int main(int argc, char** argv)
{
  // Initialise the values for the input parameters of the silent mode
  double lambda_min = static_cast<double>(-1/E);
  double lambda_max = static_cast<double>(1.0/2.0);

  // Instantiate MTQR's primary module in silent mode
  std::vector<std::vector<double>> array = mtqr(lambda_min, lambda_max);

  return 0;
}
```

The above execution of MTQR in silent mode suppresses any on-screen printing of the results and generates a single outputs.txt file containing all the information regarding the monomial transformation quadrature rule (namely the values of $\lambda_{\min}$ and $\lambda_{\max}$ and the a-posteriori relative error) followed by the new nodes and weights returned to memory. In case of previous instance of MTQR in silent mode, the exported date will be appended in outputs.txt if not deleted. Moreover, we emphasize that, though the new nodes and weights written to outputs.txt are optimised by MTQR's routine to be the proper f.p. format, those that are instead streamed to memory by the silent mode are, by default, returned in double precision regardless of the associated accuracy of the numerical integral. The reason for this choice of implementation for the silent mode of MTQR is that, rather than a placeholder type specifier (i.e. auto) to be deduced at compile time for exporting in memory the new nodes and weights in an optimised f.p. format we simply resort to a *matrix*-like structure in double precision. This choice was made to facilitate the interface of the user with our library as the double f.p. format is much more wide-spread in use than the float128 quadruple precision f.p. counterpart. Furthermore we realised that in most applications of interest in computational engineering it is frequent to only care about the extreme bounds of the input generalised polynomials, thus disregarding both all the real-valued exponents in between $\lambda_{\min} > -1$ and $\lambda_{\max}$ and also the values of the coefficients of each monomial term. As such, a study of the convergence properties of the application are of particular importance we encourage the user in instantiating the loud mode of mtqr as it always assures the optimal f.p. format for the quadrature nodes and weights. If convergence/testing analysis is not a concern, then the silent mode should be used for its enhanced efficiency.

3.4 MODULAR CALLS FOR CUSTOMISED INTEGRATION

In this section we explore in more details the methods that interact directly within the primary module mtqr. The purpose of this brief section is to expose the fundamental modular calls that are required at run-time for the execution of the monomial transformation quadrature rule implemented by the library and hidden behind the access point. By explicitly outline the scope and I/O data structures handled at run-time we intend to provide the user with a capability of using some, or all of those methods independently from the execution or integration of our library. The user can find the types that have to be defined to correctly instantiate each of the following methods. With reference to the snippet reported in Section 2.1, the mtqr.cpp source file instantiates 5 of the 12 total methods that constitute the library, which are specified in the following list sorted by the order of instantiation:

- **format_data** = manageData (**muntz_sequence**, **coeff_sequence**)
  Interfaces with user's inputs provided in the main of the application by arranging the proper data-structures.

  - **INPUTS** :

    - std::vector<T> **muntz_sequence** is a list of real numbers $\{\lambda_0, \lambda_1, \ldots, \lambda_k\}$ representing the terms of a truncated Müntz sequence of real-valued exponents associated to the generalised polynomial $p(x) \in \mathbb{P}_{\lambda_{\max}}$, $\lambda_{\max} = \max_k\{\lambda_0, \lambda_1, \ldots, \lambda_k\}$ that the user wants to integrate using the monomial transformation quadrature rule. The data type T templatising the structure is one amongst the discussed f.p. formats float128 or double;

- std::vector<T> **coeff_sequence** is a list of real numbers $\{c_{\lambda_0}, c_{\lambda_1}, \ldots, c_{\lambda_k}\}$ representing the coefficients of the terms of the generalised polynomial. The order of the elements and data type T of the structure has to match that of **muntz_sequence**.

■ **OUTPUTS** :

- std::tuple<int, std::vector<float128>> **format_data** contains an int variable $n_{\min}$ and a 2−dimensional std::vector<float128> array $\{\lambda_{\min}, \lambda_{\max}\}$; the former is the minimum number of quadrature nodes computed as the sole real root of (1.18) and the latter is the infimum and supremum of the **muntz_sequence** input list.

▶ **monomial_data** = streamMonMapData (**n_min**)
Computes and extracts all the necessary parameters to uniquely specify and build the monomial map, i.e. its order r.

■ **INPUTS** :

- int **n_min** is one of the outputs of manageData , specifically $n_{\min}$, which is easily accessed with std::get<0> (**format_data**).

■ **OUTPUTS** :

- std::tuple<int, std::vector<float128>> **monomial_data** contains an int variable $n_{\min}$ and a 2−dimensional std::vector<float128> array $\{\beta_{\min}, \beta_{\max}\}$; the former is the closest greater even integer of the minimum number of quadrature nodes provided with **n_min** while the latter is a 2−dimensional array containing the entries of BETAS.csv for the minimum and maximum monomial exponents integrated exactly with $n_{\min}$ quadrature nodes.

▶ **transf_order** = computeMapOrder (**lambdas**, **betas**)
Using the parameters streamed by the above method, compute the transformation order r of the monomial map for the input generalised polynomial.

■ **INPUTS** :

- std::vector<float128> **lambdas** is one of the outputs of manageData , specifically $\{\lambda_{\min}, \lambda_{\max}\}$ and they are accessed with std::get<0> (**format_data**);
- std::vector<float128> **betas** is one of the outputs of streamMonMapData , specifically $\{\beta_{\min}, \beta_{\max}\}$ and they are accessed with std::get<0> (**monomial_data**).

■ **OUTPUTS** :

- double **transf_order** is the real-valued transformation order $r \in \mathbb{R}$ of the monomial map introduced in (1.16).

▶ **quad_data** = computeQuadParams (**transf_order**, **n_min_even**)
Based on the monomial map computed at the previous step (specified by its order r) derives the computational parameters, i.e. nodes and weights, of the monomial transformation quadrature rule.

■ **INPUTS** :

- double **transf_order** is the output of the above computeMapOrder method;
- int **n_min_even** is one of the output of streamMonMapData , specifically n_{min}.

■ **OUTPUTS** :

- std::tuple<std::vector<float128>, std::vector<float128>, std::vector<float128>, std::vector<float128>> **quad_data** is a list of 4 vectors, namely:
  - ⋆ $\{x_1, \ldots, x_{n_{\min}}\}$ is the list of the new samples of the monomial transformation quadrature rule obtained with (1.16);
  - ⋆ $\{w_1, \ldots, w_{n_{\min}}\}$ is the list of the new weights of the monomial transformation quadrature rule obtained with (1.17);
  - ⋆ $\{\tilde{x}_1, \ldots, \tilde{x}_{n_{\min}}\}$ is the list of the samples of the classical G-L quadrature rule affinely mapped in $(0, 1)$ obtained in (1.15);

⋆ $\{\tilde{w}_1, \ldots, \tilde{w}_{n_{\min}}\}$ is the list of the weights of the classical G-L quadrature rule obtained in (1.15);.

► optimiseData (**quad_data**, **muntz_sequence**, **coeff_sequence**)

This method optimises the f.p. format (float128 or double) of the monomial transformation quadrature rules' parameters by computing the numerical integral in (1.14) and the associated a-posteriori relative error in (1.7), that are displayed at run-time.

■ **INPUTS** :

- std::tuple<std::vector<float128>, std::vector<float128>, std::vector<float128>, std::vector<float128>> **quad_data** is the output of the above computeParamsGl ;
- std::vector<T> **muntz_sequence** is the same input of method manageData i.e. the list of real-valued exponents of the input generalised polynomial with data type T that can be one of the f.p. format float128 and double ;
- std::vector<T> **coeff_sequence** is the same input of method manageData i.e. the list of real-valued coefficients of the terms of the input generalised polynomial. Data type T has to match that of the **coeff_sequence** .

■ **OUTPUTS** :

- the method does not return any data. Instead the results of the numerical integration are printed on the terminal at run-time while the new (optimised) monomial transformation quadrature rule's parameters are written on appropriate files (see Section 3.2).

3.5 ACKNOWLEDGEMENTS

This study was supported by the Italian Ministry of University and Research (MIUR) under the PRIN2017 Grant (2017NT5W7Z) *"Chipless radio frequency identification (RFID) for GREEN TAGging and Sensing - GREEN TAGS"*

Bibliography

[1] Milton Abramowitz. Handbook of Mathematical Functions, With Formulas, Graphs, and Mathematical Tables, USA: Dover Publications, Inc., 1974. ISBN: 0486612724.

[2] Boost C++ Libraries. URL: https://www.boost.org/.

[3] Boost Version 1.81.0 release. URL: https://www.boost.org/users/history/version_1_81_0.html.

[4] CMake, Kitware Inc. URL: https://cmake.org/download/.

[5] W. Gautschi. "Algorithm 726: ORTHPOL–a Package of Routines for Generating Orthogonal Polynomials and Gauss-Type Quadrature Rules". In: ACM Trans. Math. Softw. 20.1 (1994), pp. 21–62. ISSN: 0098-3500. DOI: 10.1145/174603.174605.

[6] GNU Scientific Library. URL: https://www.gnu.org/software/gsl/.

[7] R.D. Graglia and G. Lombardi. "Singular higher order complete vector bases for finite methods". In: IEEE Transactions on Antennas and Propagation 52.7 (2004), pp. 1672–1685. DOI: 10.1109/TAP.2004.831292.

[8] R.D. Graglia and G. Lombardi. "Machine Precision Evaluation of Singular and Nearly Singular Potential Integrals by Use of Gauss Quadrature Formulas for Rational Functions". In: IEEE Transactions on Antennas and Propagation 56.4 (2008), pp. 981–998. DOI: 10.1109/TAP.2008.919181.

[9] N. Hale and A. Townsend. "Fast and Accurate Computation of Gauss–Legendre and Gauss–Jacobi Quadrature Nodes and Weights". In: SIAM Journal on Scientific Computing 35.2 (2013), A652–A674. DOI: 10.1137/120889873.

[10] "IEEE Standard for Floating-Point Arithmetic". In: IEEE Std 754-2019 (Revision of IEEE 754-2008) (2019), pp. 1–84. DOI: 10.1109/IEEESTD.2019.8766229.

[11] G. Lombardi. "Design of quadrature rules for Müntz and Müntz-logarithmic polynomials using monomial transformation". In: International Journal for Numerical Methods in Engineering 80.13 (2009), pp. 1687–1717. DOI: 10.1002/nme.2684.

[12] G. Lombardi and D. Papapicco. "Algorithm XXX: A C++ Monomial Transformation Quadrature Rule for High Precision Integration of Singular and Generalised Polynomials". In: ACM Trans. Math. Softw. ?? (2022), ? DOI: ?.

[13] J.D. Ma, V. Rokhlin, and S. Wandzura. "Generalized Gaussian Quadrature Rules for Systems of Arbitrary Functions". en. In: SIAM Journal on Numerical Analysis 33.3 (June 1996), pp. 971–996. ISSN: 0036-1429, 1095-7170. DOI: 10.1137/0733048.

[14] G.V. Milovanović, T.S. Igić, and D. Turnić. "Generalized quadrature rules of Gaussian type for numerical evaluation of singular integrals". In: Journal of Computational and Applied Mathematics 278 (2015), pp. 306–325. ISSN: 0377-0427. DOI: 10.1016/j.cam.2014.10.009.