QUASIMONT

QUAdrature of SINgular polynomials using a MONonomial Transformation rule

USER MANUAL

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 the open-source C++ library QUASIMONT. Users of the library can refer to the following content for setting and executing the code as a stand alone package generating specialised quadrature rules for sets of generalised polynomials or by integrating it in their custom application for more flexibility as additional codes or static library. The principles behind the implementation of the library are discussed in the first chapter, followed by instructions for correctly building and running the library in a Linux distribution (outlined in the second and third chapter). The concluding chapter describes its inner workings and implementation. Being a mathematical software, a solid grasp of various topics in mathematical and numerical analysis is an advantage for the user to exploit the library to its fullest capability. In the second section of this chapter we provide a brief coverage on some of those concepts such as interpolatory quadrature formulae, Gaussian quadrature and asymptotic error estimation which are required for users that are not familiar with the aforementioned topics. Those are followed by the introduction of monomial transformation and its properties for quadrature of singular functions. 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 QUASIMONT.

1.1 OVERVIEW

As the name suggests, QUASIMONT provides a framework for the high-precision numerical computation of singular (definite) integrals whose integrands are generalised polynomials, i.e. whose monomial terms have non-integer degree. 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. We wrote QUASIMONT with the aim of helping the user when both generalized singular polynomials and high precision quadrature are a requirement (examples are outlined in the opening section of [12]) thus avoiding to compromise between accuracy and time of execution. The algorithm that stands at the core of the library is based on the error estimation for the Gauss-Legendre (G-L) quadrature rule applied to generalised monomials and on the ad-hoc numerical manipulation of the integration parameters (discussed in the following sections) that depends on the characteristics of the generalised polynomial 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 algorithm paper [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 for the G-L quadrature formula introduced in [11] that shows the potentialities of the G-L quadrature to integrate generalized polynomials composed by monomials of non-integer degree. Specifically the new estimate shows the same **degree of precision** of the G-L quadrature formula when applied to generalised polynomials: this 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 of non-integer degree and in particular to those characterised by an end-point singularity in the integration interval. We want to emphasize these features of QUASIMONT 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 we show in the test drivers section (see Section 2.4), QUASIMONT significantly outperforms classical G-L quadrature for polynomials whose exponents' sequence is composed of either rational or irrational numbers (henceforth referred to as **Müntz sequence** , see Subsection 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 following Chapter):

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 family of selected generalized polynomials.

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]

Whilst QUASIMONT allows for the integration of 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 will 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 [3] or may result in extremely specialized quadratures [13, 14]. QUASIMONT 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 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), 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 possible. QUASIMONT implements methods that suit a very specific demand in numerical analysis so we used a procedural programming with modules approach in its writing (as opposed to an object-oriented paradigm) in which all the different functions interact through the `quasimont` module (see Section 2.1 and 2.2).

1.2 MATHEMATICAL BACKGROUND

Given a function $f : (a, b) \to \mathbb{R}$ 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 QUASIMONT is to compute a precise numerical approximation of $I(f)$ where $f \in \mathbb{P}_k[x]$ are polynomial functions. 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 function $\tilde{f}$ to be evaluated on the integration bounds

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

However, there are applications of numerical methods in various computational sciences and engineering where such polynomials are not explicitly defined but instead they model the behavior of a computed physical quantity (e.g. source integrals in the Boundary Element Method). In scientific computing it is also frequently encountered the further constraint of integrating functions that feature 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

We let $f : (a, b) \to \mathbb{R}$ be any function; we want to compute $I(f)$ although unfortunately its primitive is not known. One solution is to substitute the function with a polynomial interpolating some of its values at sampled point, 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 will be $n-1$ (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 terms of the **Lagrangian basis** whose generators $\ell_j(x) \in \mathbb{P}_{n-1}$ are also polynomials of the same degree $n-1$. 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**

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

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 quadratures 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 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 (non polynomial) and/or singular part of $f(x)$. Now if the weight function is not null everywhere in (a, b) and we can find an infinite sequence of classical polynomials $P := \left\{p_j(x) \in \mathbb{P}_j\right\}_{j=0,1,\dots}$ 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 $\alpha \leq 2n-1$. It is therefore easy to see why Gaussian quadrature is normally preferred over Newton-Cotes formulae for the numerical integration when dealing with regular functions. Although Gaussian quadrature allows the exact integration with respect to a limited number of weight functions, they have been generalised using refined approximations and numerical effort [3]. QUASIMONT integrates generalised polynomials which are extensively used in modelling several cases of particular relevance and interest in mathematical and scientific computing, as mentioned above. The simplest and arguably most widely used Gaussian quadrature rule is the Gauss-Legendre (G-L) formula. With the G-L quadrature rule the sequence P is made of **Legendre's polynomials** which are orthogonal with respect to the constant weight function $w(x) = 1$. Usually the reference interval to define the quadrature rule is $(a = -1\,,\ b = +1)$.

1.2.3 ASYMPTOTIC ERROR ESTIMATION

This subsection analyzes the accuracy of 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 have $E_n(f) = 0$ for $k \leq 2n-1$. In the study of the performance of a scientific numerical algorithm is often more appropriate to define and use an a-posteriori (actual) relative error associated to the approximating techniques, rather than an *absolute* one. For the specific case of the numerical integration of an arbitrary function $f(x)$ via G-L quadrature we have

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

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 author 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 monomial term $f(x) = x^\lambda$, $\lambda > -1$

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

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 possible 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 $R_n(f)$ for 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 consider a finite-arithmetic threshold that we can specify to be e.g. the machine-epsilon in double f.p. format as reported in the figure below (see constant black line). Once identified the interval $(\lambda_{\min}, \lambda_{\max})$ it coincides with the real values of the exponents of the polynomial function 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 QUASIMONT extends the integration of generalised polynomials of non-integer degree in the range $(-1, +\infty)$. 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. 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 a polynomial with r terms having at least one constituting monomial of non-integer degree are to be considered Müntz (generalised) polynomials. Our library is capable of approximating the integral of 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 G-L's quadrature 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.

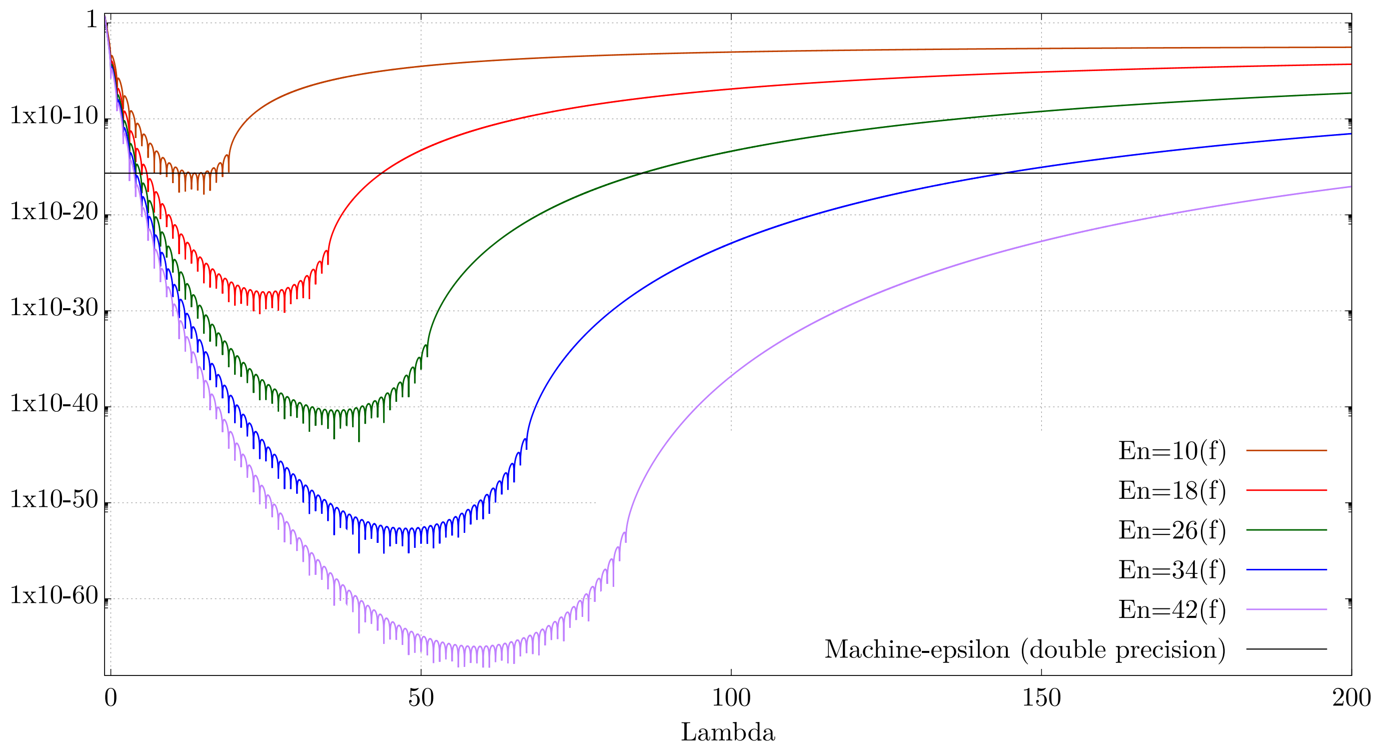

Figure 1.1: Evaluation of the relative error estimate $R_n(x^\lambda)$ provided in (1.8) for $n = 10, 18, 26, 34, 42$. We note that for a fixed n we have that the relative error is below an arbitrary (pre-determined) precision threshold in $\forall\lambda \in (\lambda_{min}, \lambda_{max})$. In the figure the solid horizontal line shows f.p. double precision. We also observe that as long as λ is a natural integer value lower than $2n-1$, then the relative error shows negative spikes in logarithmic scales, 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 real non-integer λ. This plot has been realised using the secondary module of QUASIMONT (see Section 3.4).

1.2.5 INTEGRATION INTERVAL

As reported in the algorithm paper, we consider the problem of the quadrature of generalised polynomials $f(x)$ in the interval $(0, 1)$ with a possible endpoint singularity at $x = 0$, i.e.

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

for three reasons:

- to uniquely investigate the precision property of G-L quadrature and derive unique rules to design a monomial transformation for the numerical integration algorithm;
- to completely avoid or at least severely restrict the effects of numerical cancellation in the numerical quadrature of singular generalised polynomials;
- to be coherent with the definition of Müntz polynomials and the related 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 will 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 useful to apply the monomial transformation quadrature algorithm. 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

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

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

In this subsection we outline the fundamental ingredients of the method to build the monomial transformation quadrature rule. We refer the user to [11, 12] for detailed insights regarding this topic. Given a Müntz (generalised) polynomial we define $\lambda_{\min}$ and $\lambda_{\max}$ as the smallest and greatest values 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 degree of the polynomial. Recall also that only classical polynomials (i.e. $\lambda_j \in \mathbb{N}$) feature $E_n(f) = 0$ for G-L formulae using $n = \lceil \frac{\lambda_{\max}+1}{2} \rceil$ nodes. Unfortunately, in finite arithmetic such exact result can never be achieved, and it is limited to the machine-epsilon f.p. format that is specified. Regardless, generalised singular polynomials, s.a. those in Müntz vector space, are integrated with poor accuracy by classical G-L with a prescribed number of nodes. The study of estimate (1.8) with (1.9) allows to uniquely characterise 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 monomial functions that can be integrated with a relative error strictly smaller than a fixed finite precision computed with (1.8), see also Figure 1.1. For each pair of $(\beta_{\min}(n), \beta_{\max}(n))$ there exists a unique minimum value of n (henceforth referred to as n_{min}) that satisfies the above constraint and it is estimated in [11, 12] to coincide with the sole real root of this 7th degree polynomial

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

Once all these parameters are available, 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

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

respectively, where $\tilde{x}_j$, $\tilde{w}_j$ are the classical G-L nodes and weights defined for integration in $(0, 1)$ (i.e. post affine map). As a result, the transformed new nodes and weights are optimised for the accurate integration (up to a fixed precision threshold as explained above) of the two monomials with the minimum and maximum exponents, i.e $c_{min}\, x^{\lambda_{min}}$ and $c_{max}\, x^{\lambda_{max}}$. For instance, let us consider the following Müntz polynomial

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

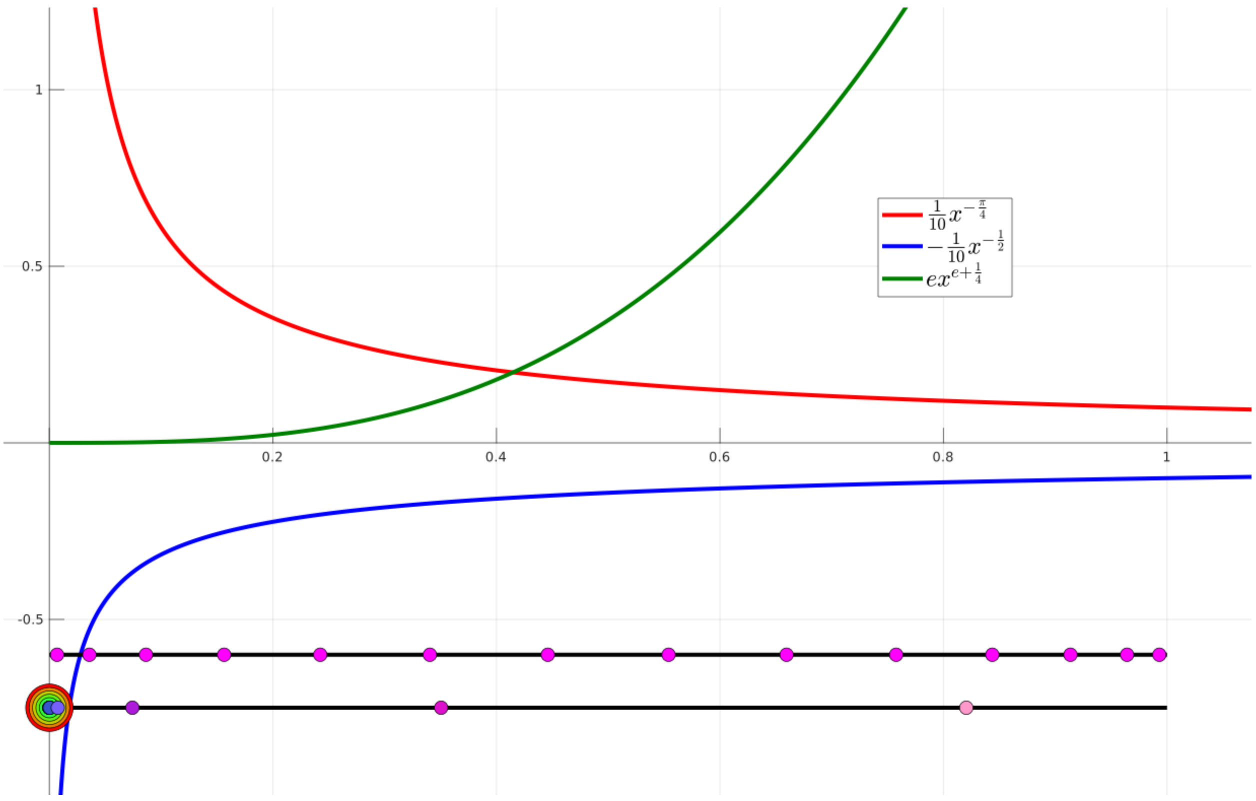

Figure 1.2: Plot of three generalised monomial terms of the Müntz polynomial $p(x)$ in (1.20). On the bottom of the graph there is the distribution in $(0, 1)$ of the $n = 14$ nodes of a classical G-L formula (magenta) and the new samples obtained by the monomial transformation quadrature rule (linear color gradient). Due to the extreme clustering of the latter (see 1.2), they are plotted with scaled-down circles as we move from $a = 0$ to $b = 1$. We clearly observe how classical G-L nodes in $(0, 1)$ are insufficient to model the singular behaviour near $x = 0$, causing a consistent loss in accuracy. On the other hand, the action of map (1.16) on the nodes (whose order $r = 28.77$ is given by (1.17)) re-arranges the vast majority of them around the endpoint singularities of $x^{-\frac{\pi}{4}}$ and $x^{-\frac{1}{2}}$, i.e. $x = 0$.

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.

	Classical G-L parameters		**Monomial quadrature (QUASIMONT)**	
$j \in \mathbb{N}$	$\tilde{x}_j \in (0, 1)$	$\tilde{w}_j \in \mathbb{R}^+$	$x_j \in (0, 1)$	$w_j \in \mathbb{R}^+$
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 $n = 14$ G-L nodes and weights before (classical) and after the application of the monomial transformation of order $r = 28.77$ depicted in Figure 1.2.

Table 1.2 reports G-L weights and points and, the resulting monomial transformation quadrature weights and points after the application of the map defined in (1.16).

Chapter 2

INSTALLATION

In the following chapter we illustrate how to build executable files with QUASIMONT from its source code as well as the dependencies needed at link-time by the resulting application. We begin from the latter by outlining the third-party source code 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 at compile and link time. Then, we address how the library is actually built and illustrate its core features through the execution of some proposed test drivers. Although all of the following content regards the execution of QUASIMONT as a stand-alone software, we postpone to the next chapter of the present manual instructions on how to use our tool as a static library fully-integrable into a user's own application.

2.1 THIRD-PARTY CODE

As reported in the opening section, QUASIMONT 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. The library that we propose is composed by two modules with corresponding `main` functions. The *primary* module is the one used to build executables and it implements all the essential methods that are required for the accomplishment of QUASIMONT's purpose, that is the computation of a monomial transformation quadrature rule that achieves double precision approximations of a definite integral whose integrands are generalized polynomials. The *secondary* module provides supporting tools for the user's in-depth understanding of the fundamental blocks of the monomial transformation. It provides the plot of the asymptotic estimate (1.8) and the computation of $\beta_{\min}(n)$ and $\beta_{\min}(n)$. As the secondary module is not essential for the correct functioning of the library (primary module), we kept it separated from the rest of the code, mainly because it introduces additional dependencies. In the following we address the main points concerning the primary module and we post-pone to Section 3.4 a brief description of the secondary module. QUASIMONT relies on a number of third-party open-source libraries. Many of them usually come shipped with C++ itself e.g. the **standard** and **vector** libraries (required for basic data-structures and functions), the **algorithm** library (needed for sorting methods), the **math** library (used for basic mathematical operations s.a. `fabs`, `pow` and `ceil`) and others. The software also requires two more libraries that are well-known in the scientific computing and open-source communities with minimum required version reported between brackets:

- **Boost C++ libraries (v-1.71)** [1]: a vast, peer-reviewed, collection of mostly header source files. In particular QUASIMONT extensively uses the `Multiprecision` library, where non-native higher precision f.p. formats have been implemented as C++ data-types. Of particular interest for our module is the IEEE 754 quadruple f.p. format which is supplied by Boost's library as *GCC's __float128* or *Intel's _Quad* data types.
- **GSL - GNU Scientific Library (v-2.5)** [6]: required by QUASIMONT 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.

These dependencies need to be installed/compiled correctly on the user's machine; furthermore macros to each static library need to be included in the PATH environment variable in order to correctly link the objects files compiled from the source code. All of the listed requirements are easily downloadable and compilable in Linux using package managers, such as the **Advanced Package Tool** and the **Yellowdog Updater, Modified**, with a single command on the terminal. Moreover the aforementioned macros should be added automatically to the PATH variable following these steps.

Installation of third-party libraries

```
user@machine: home> # For Debian-based distros (Ubuntu, Mint, Knoppix, Kali ...)
user@machine: home> sudo apt-get update -y
user@machine: home> sudo apt-get install libboost-all-dev libgsl-dev
user@machine: home> # For RPM-based distros (CentOS, Fedora, SUSE, Scientific Linux ...)
user@machine: home> sudo yum update -y
user@machine: home> sudo yum install boost-devel gsl-devel
```

Additionally, QUASIMONT requires the proper installation of the appropriate building tools; at the present moment the library has been written for Linux platforms only and therefore we prioritized a minimalist and straightforward build process over cross-platform compliance by adopting the usage of *makefile* (see Section 2.3). The default compiler we selected for building QUASIMONT is the **GCC - GNU Compiler Collection** [4] that is invoked from the **GNU Make** [5] program. For this reason the installation of gcc and make are required on the user machine. Also in this case the aforementioned package managers allow fast and simple installation of the tools with minimal input on the terminal.

Installation of building tools

```
user@machine: home> # For Debian-based distros (Ubuntu, Mint, Knoppix, Kali ...)
user@machine: home> sudo apt-get install build-essential
user@machine: home> gcc --version
user@machine: home> make --version
user@machine: home> # For RPM-based distros (CentOS, Fedora, SUSE, Scientific Linux ...)
user@machine: home> sudo yum install gcc gcc-c++ make
user@machine: home> gcc --version
user@machine: home> make --version
```

We remark that our QUASIMONT's built has been achieved using versions 9.3.0 and 4.2.1 of GCC and GNU Make respectively. On some non-Debian based distros (e.g. CentOS) the default version of gcc and boost installed by the package manager would be significantly lower than the prescribed 9.3.0 and 1.71 . We reiterate that QUASIMONT is written in C++17 which is supported on gcc by means of the compiler flag std=c++17 only from version 8.0 onward; furthermore the float128 datatype is implemented in boost from version 1.71 onward. In case the user's package manager does not allow for the direct installation of such versions of gcc and boost then they must be built from source following the instructions on their webpage. Once all the above packages and tools have been installed, the user should thoroughly check their correct configuration, having care in particular if links to the static libraries of each of the two major dependencies listed above have been added to gcc 's linker ld standard search paths. The dependencies are constituted of large libraries, although QUASIMONT 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. For the sake of completeness we hereby list all the methods of the selected third-party libraries that are used by QUASIMONT. These methods can be located in the source code, specifically in the header file Quasimont.h , reported in the snippet below. As for any other source file in the library, a comment block precedes the code content, in which the first line specifies the location of it in the library's relative directory's tree (see Figure 2.1). We note that all external source code headers are included in Quasimont.h together with the declarations of all the implemented functions (see Section 2.2).

Quasimont.h

```cpp
//--------------------------------------------------------------------------------------
// File:       include/Quasimont.h
//
// Library:    QUASIMONT-QUAdrature of SIngular polynomials using MONomial Transformations:
//                       a C++ library for high precision integration of singular
//                       polynomials of non-integer degree
//
// Authors:    Guido Lombardi, Davide Papapicco
//
// Institute: Politecnico di Torino
//             C.so Duca degli Abruzzi, 24 - Torino (TO), Italia
//             Department of Electronics and Telecommunications (DET)
//             Electromagnetic modelling and applications Research Group
//--------------------------------------------------------------------------------------

#ifndef QUASIMONT_H
#define QUASIMONT_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/float128.hpp>
#include <gsl/gsl_poly.h>

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

#include "Utils.h"  // includes header file for plotting and other utilities
#include "DatIo.h"  // includes header file for data I/O functions
#include "MonMap.h" // includes header file for functions computing the monomial map

#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

template<typename type>
void quasimont(std::vector<type>& muntz_sequence, std::vector<type>& coeff_sequence);

#endif // QUASIMONT_H
```

Further down in Quasimont.h we find definitions of some constants used throughout the library and finally the declaration of the access point of the primary module of the library, i.e. the method quasimont in which all other methods of QUASIMONT interact. The definition of such method is given in the homonym source file Quasimont.cpp reported below. On such file we highlight the comment block at the beginning of the definition of the function itself (lines 18-31) where the user finds quick useful insights about its usage, I/O and implementation. Such pattern is shared by each method across the main module of the library (see Chapter 4).

Quasimont.cpp

```
//-----------------------------------------------------------------------------------------
// File:       src/Quasimont.cpp
//
// Library:    QUASIMONT-QUAdrature of SIngular polynomials using MONomial Transformations:
//                       a C++ library for high precision integration of singular
//                       polynomials of non-integer degree
//
// Authors:    Guido Lombardi, Davide Papapicco
//
// Institute:  Politecnico di Torino
//             C.so Duca degli Abruzzi, 24 - Torino (TO), Italia
//             Department of Electronics and Telecommunications (DET)
//             Electromagnetic modelling and applications Research Group
//-----------------------------------------------------------------------------------------

#include "Quasimont.h"

//////////////////////////////////////////////////////////////////////////////////////////
//
//        FUNCTION: quasimont(muntz_sequence, coeff_sequence)
//
//           INPUT: - muntz_sequence = sequence of real exponents of the polynomial
//                  - coeff_sequence = sequence of real coefficients of the polynomial
//
//          OUTPUT: - no outputs
//
//     DESCRIPTION: access point of the primary module of the library where all the
//                  methods concerning the computation of the monomial transformation
//                  quadrature rule's params are instantiated according to user's input.
//
//////////////////////////////////////////////////////////////////////////////////////////

template<typename type>
void quasimont(std::vector<type>& muntz_sequence, std::vector<type>& coeff_sequence)
{
  // PRINT INITIAL MESSAGE AND SELECTS USER INPUTS
  auto input_data = manageData(muntz_sequence, coeff_sequence);

  // EXTRACT N_MIN, BETA_MIN AND BETA_MAX
  auto monomial_data = streamMonMapData(std::get<0>(input_data));

  // COMPUTE THE MONOMIAL TRANSFORMATION ORDER
  double transf_order = computeMapOrder(std::get<1>(input_data), std::get<1>(monomial_data)
    ↪ );

  // COMPUTE AND EXPORT THE NEW G-L NODES & WEIGHTS
  auto quad_data = computeParamsGl(transf_order, std::get<0>(monomial_data));

  // CONVERTS AND EXPORTS NEW NODES AND WEIGHTS IN THE MOST OPTIMISED FLOATING-POINT FORMAT
    ↪  POSSIBLE
  optimiseData(quad_data, muntz_sequence, coeff_sequence);
}
template void quasimont(std::vector<float128>& muntz_sequence, std::vector<float128>&
    ↪ coeff_sequence);
template void quasimont(std::vector<double>& muntz_sequence, std::vector<double>&
    ↪ coeff_sequence);
```

2.2 STRUCTURE

The source-code in the library does not use relative paths for finding the definitions of its methods in the headers; relative paths are instead used at compile and link time by the makefile (see Section 2.3). The user is nonetheless discouraged from moving files and/or changing those paths because they do appear occasionally in the source code for retrieving data from specific non-source files. It is important therefore to understand where these files are stored and how the code is organised. As mentioned in the previous chapters, QUASIMONT is not object-oriented and all its methods interact through the access point of the primary module quasimont that interfaces the user through the inputs defined in the `main` function. Apart from it, all the remaining source code that constitutes the primary module of the proposed software is made of only 12 methods whose definitions are collected in one of the following three source files and related headers:

- `MonMap.cpp` contains every method associated with the computation of the monomial transformation quadrature rule, ranging from the monomial map itself (i.e. $\beta_{\text{min/max}}$ 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

MonMap.h

```cpp
//------------------------------------------------------------------------------------------
// File:       include/MonMap.h
//
// Library:    QUASIMONT-QUAdrature of SIngular polynomials using MONomial Transformations:
//                       a C++ library for high precision integration of singular
//                       polynomials of non-integer degree
//
// Authors:    Guido Lombardi, Davide Papapicco
//
// Institute: Politecnico di Torino
//             C.so Duca degli Abruzzi, 24 - Torino (TO), Italia
//             Department of Electronics and Telecommunications (DET)
//             Electromagnetic modelling and applications Research Group
//------------------------------------------------------------------------------------------

#ifndef MONMAP_H
#define MONMAP_H

// (SEE LINES 18~39 IN 'src/MonMap.cpp') 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);

// (SEE LINES 55~75 IN 'src/MonMap.cpp') 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);

// (SEE LINES 117~132 IN 'src/MonMap.cpp') 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);

// (SEE LINES 176~195 IN 'src/MonMap.cpp') Computes the new nodes and weights of the G-L
    ↪ formula
std::tuple<std::vector<float128>, std::vector<float128>, std::vector<float128>, std::vector
    ↪ <float128>> computeParamsGl(const double& r, const int& n_min);

// (SEE LINES 298~324 IN 'src/MonMap.cpp') Computes the numerical integral with G-L
    ↪ quadrature formula
template<typename type1, typename type2>
float128 computeQuadGl(const std::vector<type1>& nodes, const std::vector<type1>& weights,
    ↪ std::vector<type2>& muntz_sequence, std::vector<type2>& coeff_sequence);

// (SEE LINES 352~374 IN 'src/MonMap.cpp') Computes the a-posteriori relative error of the
    ↪ G-L quadrature using the new nodes and weights
template<typename type>
float128 computeExactError(const float128& In, std::vector<type>& muntz_sequence, std::
    ↪ vector<type>& coeff_sequence, bool& print_primitive);

#endif // MONMAP_H
```

- DatIo.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 corresponding header file

DatIo.h

```cpp
//------------------------------------------------------------------------------------------
// File:      include/DatIo.h
//
// Library:   QUASIMONT-QUAdrature of SIngular polynomials using MONomial Transformations:
//                      a C++ library for high precision integration of singular
//                      polynomials of non-integer degree
//
// Authors:   Guido Lombardi, Davide Papapicco
//
// Institute: Politecnico di Torino
//            C.so Duca degli Abruzzi, 24 - Torino (TO), Italia
//            Department of Electronics and Telecommunications (DET)
//            Electromagnetic modelling and applications Research Group
//------------------------------------------------------------------------------------------

#ifndef DATIO_H
#define DATIO_H

// (SEE LINES 18~46 IN 'src/DatIo.cpp') 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);

// (SEE LINES 208~226 IN 'src/DatIo.cpp') Extract the values of beta_min and beta_max
    ↪ according to the computed minimum number of nodes
std::tuple<int, std::vector<float128>> streamMonMapData(const int& comp_num_nodes);

// (SEE LINES 268~292 IN 'src/DatIo.cpp') Degrade the precision of the new G-L 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);

// (SEE LINES 382~410 IN 'src/DatIo.cpp') Computes and exports the resulting G-L weights
    ↪ and nodes aling 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 // DATIO_H
```

- VecOps.cpp defines every function that does not perform either direct computations for the quadrature rule nor it performs I/O operations on data. The methods defined in it are used to automatise specific operations on vectors that are required multiple times across the library. castVector method for example converts vectors' values in *higher-than-double* floating-point format via static_cast of each of its elements, while doubleDotProduct implements the inner product operation between two vectors with high precision avoiding numerical cancellations of smaller-than-epsilon values. Given the generic nature of their task, no naming scheme is assigned to these methods whose declarations are found in the header file Utils.h below. In Utils.h we also find declarations of the 3 methods of the secondary module that are defined, alongside its main, in the ErrTools.cpp source file (see Section 3.4).

Utils.h

```cpp
//-----------------------------------------------------------------------------------
// File:       include/Utils.h
//
// Library:    QUASIMONT-QUAdrature of SIngular polynomials using MONomial Transformations:
//                       a C++ library for high precision integration of singular
//                       polynomials of non-integer degree
//
// Authors:    Guido Lombardi, Davide Papapicco
//
// Institute: Politecnico di Torino
//             C.so Duca degli Abruzzi, 24 - Torino (TO), Italia
//             Department of Electronics and Telecommunications (DET)
//             Electromagnetic modelling and applications Research Group
//-----------------------------------------------------------------------------------

#ifndef UTILS_H
#define UTILS_H

// (SEE LINES 18~30 IN 'utilities/VecOps.cpp') Returns the float128 input vector in a type
    ↪ specified by the instatiantiation
template<typename type>
std::vector<type> castVector(const std::vector<float128>& input_vector, const type&
    ↪ type_infer);

// (SEE LINES 45~61 IN 'utilities/VecOps.cpp') 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);

// (SEE LINES 33~47 IN 'utilities/ErrTools.cpp') Generates n equispaced points between two
    ↪ input real numbers
template<typename type>
std::vector<type> linspacedVector(const type& start_type, const type& end_type, const int&
    ↪ num_steps);

// (SEE LINES 77~90 IN 'utilities/ErrTools.cpp')
template<typename type>
type aPrioriAsympEstimate(const type& input_lambda, const int& num_nodes);

// (SEE LINES 114~129 IN 'utilities/ErrTools.cpp')
template<typename type>
void plot(const int& num_nodes, const type& beta_min, const type& beta_max);

// (SEE LINES 167~180 IN 'utilities/ErrTools.cpp')
void printProgressBar(const int& iter, const int& num_iter);

#endif // UTILS_H
```

With reference to Figure 2.1 we present a visual sketch of the files' organisation in the library's directory. Source files of the primary module, i.e. MonMap.cpp and DatIo.cpp are located in the subdirectory **QUASIMONT/src** alongside Quasimont.cpp. On the other hand the VecOps.cpp source file is instead placed in the **QUASIMONT/utilities** subdirectory where we can also find the secondary module's source code ErrTools.cpp. Despite such separation, all the header files outlined above are located in the **QUASIMONT/include** subdirectory, including Quasimont.h. We recall that some relative paths are essential for the correct loading of raw data from **tabulated data-files** into the source code.

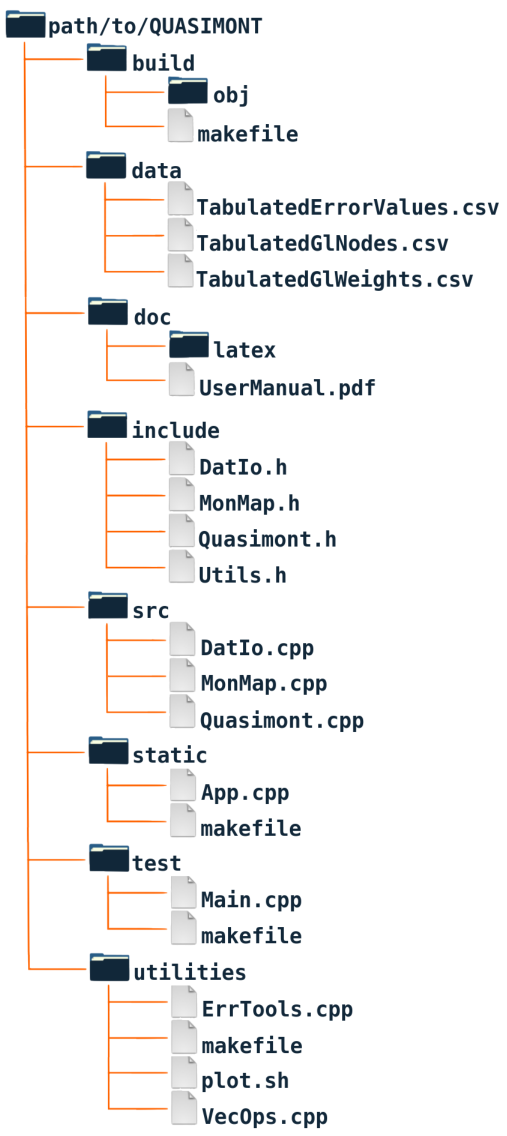

Figure 2.1: The directory tree representing the structure and organisation of QUASIMONT

Those files are:

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

We note that the latter two tabulated values of the original G-L nodes and weights in $(-1, 1)$ are stored with 50 decimal digits of precision. Their automatic computation is outside the scope of the library and already explored in details and implemented in other works [3, 9]. Those files are located in **QUASIMONT/data** and shall never be edited. In the **QUASIMONT/build** the makefile used for building the library and its applications is located whereas **QUASIMONT/build/obj** subdirectory contains the object files created at compile-time. Finally, test drivers (see Section 2.4) used for checking the proper installation and execution of the application are located in **QUASIMONT/test** while in **QUASIMONT/static** we stored an example application for compiling and using the static library version of our software (see Sub-Section 3.3).

2.3 BUILD PROCESS

QUASIMONT is a quadrature tool that can be easily integrated in users' applications, while creating executable. In the provided library, the integration is performed, for example, through the compilation and linking of the library's source code with a `Main.cpp` file containing the user's input, thus called **input source file** (see Section 3.1). This file determines the *gateway* for the access point `quasimont` of the primary module and therefore there should be one in any application built by the user. This is due to a fundamental rule in C++, for which only one `main` is defined at global level. By design, in our library, `quasimont` method acts as the designated *entry point* of the monomial transformation quadrature algorithm at run-time.

Application's own (top-level) makefile

```
#-------------------------------------------------------------------------------------
# File:       test/makefile
#
# Library:    QUASIMONT-QUAdrature of SIngular polynomials using MONomial Transformations:
#                  a C++ library for high precision integration of singular
#                  polynomials of non-integer degree
#
# Authors:    Guido Lombardi, Davide Papapicco
#
# Institute:  Politecnico di Torino
#             C.so Duca degli Abruzzi, 24 - Torino (TO), Italia
#             Department of Electronics and Telecommunications (DET)
#             Electromagnetic modelling and applications Research Group
#-------------------------------------------------------------------------------------

CXX = g++

BLDDIR = ../build
INCDIR = ../include
SRCDIR = ../src
UTLDIR = ../utilities
OBJDIR = $(BLDDIR)/obj

OBJ = $(OBJDIR)/Main.o $(OBJDIR)/Quasimont.o $(OBJDIR)/VecOps.o

CXXOPTIONS = -ansi -std=c++17
CXXFLAGS = $(CXXOPTIONS) -I $(INCDIR)

LDLIBS = -lm -lquadmath -lgsl -lgslcblas
LDFLAGS = $(LDLIBS)

default: $(OBJ)
  $(CXX) $(OBJDIR)/*.o -o Test $(LDFLAGS)
  ./Test

$(OBJDIR)/Main.o: Main.cpp
  $(CXX) -c $< -o $@ $(CXXFLAGS)

$(OBJDIR)/Quasimont.o:
  cd $(BLDDIR) && $(MAKE)

$(OBJDIR)/VecOps.o:
  cd $(UTLDIR) && $(MAKE) $@

tools:
  cd $(UTLDIR) && $(MAKE) $@

clean:
  rm -f $(OBJDIR)/Main.o Test
  if [ -d "output" ]; then rm -r output; fi

allclean:
  $(MAKE) clean
  cd $(UTLDIR) && $(MAKE) clean
  cd $(BLDDIR) && $(MAKE) clean
```

The source code of the library itself (i.e. the source files in **QUASIMONT/src**) does not define any main function and thus can only be compiled but not executed by itself. In order to build applications, the compiled source code has to be linked with a compiled main that provides the access point of the primary module to the program's executable. We can define a sort of one-to-one correspondance between each application's executable and its associated main function, the latter uniquely identified by the input singular generalised polynomials defined by the user. The build process is based on a *recursive make* approach, i.e. the input source file is built through a top-level makefile (reported above) that recursively invokes the makefile responsible for building the source code of the library, its utilities and third-party dependencies. The library source code shall be compiled just once and the resulting objects shall remain available thereafter for linking to any main , i.e. new applications. Whenever the rule allclean is invoked from the top-level makefile , it will delete those object files and the source code will require to be compiled again. This practice is useful for users who require to execute standalone applications by reusing the unchanged compiled objects of the source code multiple times. In these cases we remark that the output consists of the specialised quadrature samples in addiction to the numerical estimation of the integral for a generalised polynomial. Such output data is stored in a specific subdirectory as described at Section 3.2.

Compilation and Linking of the library

```
# First clean the object files from any previous compilation
user@machine: home/QUASIMONT/test> make allclean
make clean
make[1]: Entering directory '/home/QUASIMONT/test'
rm -f ../build/obj/Main.o Test
if [ -d "output" ]; then rm -r output; fi
make[1]: Leaving directory '/home/QUASIMONT/test'
cd ../utilities && make clean
make[1]: Entering directory '/home/QUASIMONT/utilities'
rm -f Tools && if [ -d "estimate" ]; then rm -r estimate; fi
rm -f ../build/obj/*.o
make[1]: Leaving directory '/home/QUASIMONT/utilities'
cd ../build && make clean
make[1]: Entering directory '/home/QUASIMONT/build'
rm -f obj/*.o
make[1]: Leaving directory '/home/QUASIMONT/build'
# Then the build starts by compiling the library source into object files
user@machine: home/QUASIMONT/test> make
g++ -c Main.cpp -o ../build/obj/Main.o -ansi -std=c++17 -I ../include
cd ../build && make
make[1]: Entering directory '/home/QUASIMONT/build'
g++ -c ../src/Quasimont.cpp -o obj/Quasimont.o -g -ansi -std=c++17 -I ../include
g++ -c ../src/DatIo.cpp -o obj/DatIo.o -g -ansi -std=c++17 -I ../include
g++ -c ../src/MonMap.cpp -o obj/MonMap.o -g -ansi -std=c++17 -I ../include
make[1]: Leaving directory '/home/QUASIMONT/build'
cd ../utilities && make ../build/obj/VecOps.o
make[1]: Entering directory '/home/QUASIMONT/utilities'
g++ -c VecOps.cpp -o ../build/obj/VecOps.o -g -ansi -std=c++17 -I ../include
make[1]: Leaving directory '/home/QUASIMONT/utilities'
# Once all the objects have been compiled the linker automatically generates the executable
    ↪ and launches the simulation
g++ ../build/obj/*.o -o Test -lm -lquadmath -lgsl -lgslcblas
./Test

    |-------------------------------------------------|
    |                 ** QUASIMONT **                 |
    |  ** MONOMIAL TRANSFORMATION QUADRATURE RULE **  |
    |-------------------------------------------------|


 Input polynomial p(x) = ...
```

On the other hand, if integration of QUASIMONT in larger more complex software is required (e.g. in FEM/BEM libraries), linking to the static library is necessary, which is addressed later in the manual at Section 3.3. To conclude the discussion on how the QUASIMONT is built locally, let us display the outcome of some of the basic rules defined in the top-level makefile . The output of the compilation of the source files are object files that will be stored in the **QUASIMONT/build/obj** subdirectory. The linking of those objects is invoked only by the top-level makefile associated to the specific application. For each application built by the user it is considered a good practice to invoke the clean rule prior to the build process itself. We emphasize that, with the exception of those cases when QUASIMONT is used as a tool in a larger library, each application should be associated to its own subdirectory located within QUASIMONT itself. In this subdirectory a user input source file containing the main, named Main.cpp by default, should be located. User may opt for changing the filename of main as it does not impact any aspect of the source code as long as the same name appears in the default rule reported in the top-level makefile of the application, which should also be located in the subdirectory. For simplicity, we suggest to retain the naming convention reported in the provided makefile. In this scheme, the application's name (in our example Test) matches that of the directory in which its input source file Main.cpp is located (i.e. **QUASIMONT/test**). In the next subsection a reference application located in **QUASIMONT/test** (already shipped with the library) exemplifies the whole build sequence.

2.4 INSTALLATION AND TEST DRIVER

The installation of the library can be done by building the Main.cpp input source file in the **QUASIMONT/test** subdirectory, which contains the test driver for benchmarking QUASIMONT's performance. The Test application runs the library's processing methods onto three generalised and classical polynomial integrands that introduce the user to the library interaction, execution and output whilst assuring that the core functionalities have been compiled correctly (no conflict at linking stage occurs). 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 $x = 0$. We set the integration interval to be $(0, 1)$ by default in QUASIMONT. Once both the library's and user input's source files have been compiled and linked into the corresponding executable, the resulting application integrates each polynomial using the monomial transformation quadrature rule. A list of important information regarding both the I/O and 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). To check the correctness of the built application, the user should verify that the results obtained do indeed match the output data reported in the following tables 2.1, 2.2, 2.3 for each polynomial function. To run each benchmark the user needs to return to the CLI where the program is waiting for her/him to go ahead to the next case. 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 QUASIMONT 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 the typical integrand function often encountered in numerical methods for differential and integral equations featuring singular modelling, on which QUASIMONT is indeed considered an helpful, effective and precise tool. The proposed generalised polynomial $p_1(x)$ features a strong 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 $(0, 1)$.

Building and executing the test driver: p_1(x)

```
user@machine: home/QUASIMONT/test> make clean
make clean
rm -f ../build/obj/Main.o Test
if [ -d "output" ]; then rm -r output; fi
user@machine: home/QUASIMONT/test> make
g++ -c Main.cpp -o ../build/obj/Main.o -ansi -std=c++17 -I ../include
g++ ../build/obj/*.o -o Test -lm -lquadmath -lgsl -lgslcblas
./Test

       |------------------------------------------------|
       |                  ** QUASIMONT **               |
       |  ** 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)) = 5.4523153678583704e-18  [with parameters in double precision] **

  ** QUASIMONT HAS TERMINATED **
```

Table 2.1 shows what was graphically reported in Figure 1.2. The classical G-L nodes of the associated quadrature rule with $n = 32$ are in fact distributed symmetrically in $(0, 1)$ with an equal abundance of those at both bounds of the integration interval, while the proposed quadrature based on the monomial transformation concentrates nodes towards the lower bound $x = 0$, squashing the vast majority of them (26 out of the total 32) inside the sub-interval $(0, 0.1)$ where the singularity behaviour is captured. The second benchmark polynomial is proposed with the scope of introducing the user to a core functionality of QUASIMONT 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. QUASIMONT allow 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.

Building and executing the test driver: p_2(x)

```
       |------------------------------------------------|
       |                  ** QUASIMONT **               |
       |  ** MONOMIAL TRANSFORMATION QUADRATURE RULE **  |
       |------------------------------------------------|

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

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

Nodes and weights (double f.p. format) 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	0.01598386920155067	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 2.1: 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.

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. 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 $\tilde{p_2}(x) = x^{-\frac{\epsilon}{3}} + x^{\lambda_{\max}}$ accordingly. The second path allows instead to specify $\lambda_{\max}$ directly and let QUASIMONT 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` will cause 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, will cause QUASIMONT to throw an error message and exit the program. After the specified selection, QUASIMONT 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 direct the user's attention to the fact that the latter format has been selected despite being slightly less precise than the former. The logic behind is that both relative errors are compute at run-time and if the absolute difference is smaller than twice the machine-epsilon in double precision than we go ahead and export the data in the optimised double format.

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] **

 ** QUASIMONT HAS TERMINATED **
```

Nodes and weights (double f.p. format) 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	7.80153953804910668e-95	1.8746032945680622e-93	36	2.8538360103496875e-11	5.3006402608938980e-11
5	1.37705792986285616e-85	2.6096113660080927e-84	37	1.7474698043212560e-10	3.0884893242340361e-10
6	3.99955200879667989e-78	6.2503681667652076e-77	38	9.7966474232194539e-10	1.6467041924026613e-09
7	7.12412148203568704e-72	9.4622687681839969e-71	39	5.0450425037572576e-09	8.0595541550951593e-09
8	1.67758621116242340e-66	1.9352874634013771e-65	40	2.3937780380919019e-08	3.6314704984080236e-08
9	8.50394804246417862e-62	8.6610446966327004e-61	41	1.0493924006829552e-07	1.5103117353143750e-07
10	1.28937420430520282e-57	1.17417554134314e-56	42	4.2611947777521824e-07	5.8115902014625234e-07
11	7.37695543314284930e-54	6.0678528535818764e-53	43	1.6064924620227659e-06	2.0734907170162103e-06
12	1.88823854215581744e-50	1.4144870181471483e-49	44	5.6352337753679891e-06	6.8727595063813297e-06
13	2.45874041791221323e-47	1.6888846224093984e-46	45	1.8428406102394049e-05	2.1200152242057246e-05
14	1.79879095499131601e-44	1.1394463566176748e-43	46	5.6285102084355121e-05	6.0953635295967037e-05
15	7.99644755427870559e-42	4.6939290576395100e-41	47	0.000160824501490444	0.00016357104996692
16	2.30031834550869425e-39	1.256455692921632e-38	48	0.000430554366218858	0.000410178545783102
17	4.50736224431423266e-37	2.2990244539015347e-36	49	0.001081507980701627	0.00096212522184789
18	6.27605017641916653e-35	2.9984883056622040e-34	50	0.002552202028188329	0.002112665166000562
19	6.43329661193267433e-33	2.8866958399889620e-32	51	0.005664918590851289	0.004345383033450631
20	5.00167986643949820e-31	2.1127162036193374e-30	52	0.011839418057758923	0.008374932123081819
21	3.02522544694279073e-29	1.2053618210763572e-28	53	0.023321020012667499	0.015126268661501289
22	1.45485132139284000e-27	5.4774213217611214e-27	54	0.043333892814950261	0.025596514741963807
23	5.66857779784746315e-26	2.0197293557945103e-25	55	0.076017823001042698	0.040555711474099192
24	1.81901613177956244e-24	6.1417478949789698e-24	56	0.1259851142914331	0.060094184332119245
25	4.87674410785629622e-23	1.5621281667244454e-22	57	0.197384568387222875	0.083114399815460055
26	1.10619188116815056e-21	3.3648881412038240e-21	58	0.292508401834271963	0.106964653637029898
27	2.14674169704207817e-20	6.2062442929374053e-20	59	0.410207778274644215	0.127463313674079491
28	3.59979724510255679e-19	9.8975903331295861e-19	60	0.544613347654060773	0.139504287218888873
29	5.26211017870698955e-18	1.3767307083491291e-17	61	0.68476273369408191	0.138243738177407818
30	6.75862034692174972e-17	1.68331269687316e-16	62	0.815603424165963892	0.120596149243306058
31	7.68166812405570700e-16	1.8218265936883635e-15	63	0.920443566774952144	0.086540144565319482
32	7.77553771704852579e-15	1.756314510825820e-14	64	0.984393322175774548	0.039739900698273792

Table 2.2: 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.

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 QUASIMONT. According to the properties described in Section 1.2.2, the former requires $n = \frac{\lambda_{max}+1}{2} = 18$ nodes to achieve double precise integration.

Building and executing the test driver: p_3(x)

```
    |--------------------------------------------------|
    |                  ** QUASIMONT **                 |
    |  ** 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))    = 0.0833333333333333333 **
 ** I_n(p(x)) = 0.0833333333333333326    [with parameters in float128 precision] **
 ** E_n(p(x)) = 8.8727918916220005e-17   [with parameters in float128 precision] **
 ** I_n(p(x)) = 0.0833333333333333323    [with parameters in double precision] **
 ** E_n(p(x)) = 1.1803389750004119e-16   [with parameters in double precision] **

 ** QUASIMONT HAS TERMINATED **

user@machine: home/QUASIMONT/test>
```

While running QUASIMONT, we immediately note that our proposed software outperforms G-L quadrature for double precision numerical integration 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 generalised polynomials with $\lambda \in \{17, 35\}$ with machine-epsilon precision specified for the double f.p. format, as it can be assessed by the table below.

Nodes and weights (double f.p. format) 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 2.3: 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.

With these results we would like to direct the user's attention on the possibility of using QUASIMONT 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 $\lambda_{\text{min}} > -1$ holds for the integrand. By excluding those cases of rational functions we argue that the usage of QUASIMONT 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.

Chapter 3

USER INTERFACE

The execution of the test drivers should have introduced the user on the fundamentals and interaction with the library. In this chapter we will therefore expose the I/O interface and how the user can quickly build a new application. In particular a step-by-step procedure to setup, build and execute a custom-made application from scratch will be described. In the first section we discuss the input source file, amendable by the user to create her/his first application with the library. In the second section we address how the results are exported. Finally we briefly outline how these results can be used by other applications and loaded in external code automatically by integrating QUASIMONT in a larger library and conclude with a short description of the previously mentioned secondary module.

3.1 THE INPUT SOURCE FILE

Let us start by implementing a new exemplifying application by reconsidering the singular generalised polynomial $p_1(x)$ introduced in the test driver of Sub-Section 2.4. We begin by creating a copy of the source file Main.cpp and of the associated top-level makefile of test application located in **QUASIMONT/test** into a new subdirectory labelled **QUASIMONT/MyApp**

Creation of application's directory

```
user@machine: home/QUASIMONT> cp -r test MyApp
user@machine: home/QUASIMONT> cd MyApp/
user@machine: home/QUASIMONT/MyApp> make clean
rm -f ../build/obj/Main.o Test
if [ -d "output" ]; then rm -r output; fi
```

In both Main.cpp and makefile we change the inputs according to the new application. We start from the latter for which the amendments amount to

- substitute the string Test to MyApp whenever it is mentioned to effectively rename the application built by the compiler;
- alternatively at line define a macro variable EXE_NAME = MyApp and invoke it as \$(EXE_NAME) at the corresponding lines above.

Recalling that QUASIMONT is designed to perform precise numerical integration of generalised polynomials defined in the interval $(0, 1)$, we need to modify the inputs related to the two sequences of coefficients and exponents that uniquely define the polynomial itself in Main.cpp, i.e.

- the coefficients_sequence stored in a std::vector of length $r+1$, that is the number of non null-coefficients (see Sub-section 1.2.4);
- the muntz_sequence of exponents of the polynomial $p(x)$ also stored in a std::vector of the same length $r+1$

If the two data-structures have different lengths, QUASIMONT will throw an error message and exit the program. We shall retain the exact same sequence that has been previously defined in $p_1(x)$, i.e. lines 21-22

Main.cpp (input source file)

```
 1 //--------------------------------------------------------------------------------------
 2 // File:       MyApp/Main.cpp
 3 //
 4 // Library:    QUASIMONT-QUAdrature of SIngular polynomials using MONomial Transformations:
 5 //                       a C++ library for high precision integration of singular
 6 //                       polynomials of non-integer degree
 7 //
 8 // Authors:    Guido Lombardi, Davide Papapicco
 9 //
10 // Institute: Politecnico di Torino
11 //             C.so Duca degli Abruzzi, 24 - Torino (TO), Italia
12 //             Department of Electronics and Telecommunications (DET)
13 //             Electromagnetic modelling and applications Research Group
14 //--------------------------------------------------------------------------------------
15 
16 #include "Quasimont.h"
17 
18 int main(int argc, char** argv)
19 {
20   //  P(x) = ex^(e+1/4) + 5x^(pi/4) -x^(-1/2) +1 +10x^2
21   std::vector<float128> coeff_sequence = {E, 5.0, -1.0, 1.0, 10.0};
22   std::vector<float128> muntz_sequence = {E + 0.25, -PI/4, -0.5, 0, 2};
23 
24   quasimont(muntz_sequence, coeff_sequence);
25 
26   return 0;
27 }
```

Order of input of either muntz_sequence and coefficients_sequence does not matter as QUASIMONT automatically extract $\lambda_{\min}$ and $\lambda_{\max}$ through a *sorting* algorithm. Regardless of the *"absolute"* order, it is trivial that the user must make sure that the *"relative"* order of the two sequences must coincide. Once we correctly defined all the inputs we can instantiate the primary method quasimont, then save the amended file and exit the editor. Although the aim of QUASIMONT 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 missing any digit. Applications with coefficients and exponents sequences defined with double precision is straightforward.

3.2 RESULTS AND OUTPUTS

In the top-level makefile of the Test application we embedded the automatic execution of the program once the compilation is (successfully) completed. If the user wants to disable such execution, it is necessary to erase the corresponding line of the makefile and manually input ./MyApp on the CLI at the end of the compilation. Once the application is ran, 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 (see below for further explanation) and lastly the value of the primitive (or analytic integral), numerical integral and the relative error. The classical G-L parameters are stored in TabulatedGlNodes.csv and TabulatedGlWeights.csv in text format and imported 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 data, QUASIMONT features a method called optimiseData (see Sub-section 4.2.3) that automatically selects the most optimised format possible, among those listed in Section 1.1, with which to export the output, results i.e. the new quadrature nodes and weights, to reach the prescribed relative precision for the integration (e.g. double 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 are not shown on the terminal but are instead exported in three separate files called `Results.txt`, `Nodes.txt` and `Weights.txt`, all located in the **QUASIMONT/MyApp/output** subdirectory created automatically by the application. The text files introduced above collect 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 QUASIMONT. The remaining two files, as their names suggest, list the actual output of the library i.e. the new nodes and weights respectively. The user should compare her/his results with those listed in the previous Table 2.1. The resulting new quadrature samples, established by the monomial rule and optimised accordingly by the aforementioned routine, are exported with the f.p. precision guarantees the quadrature relative error to be within the machine-epsilon in double precision. The choice was made here to use text files instead of streaming them as outputs of the primary method `quasimont` so that a permament record of them will be retrievable, at any time, by the user, regardless of the type of application and software integration of our library. We emphasize that by instantiating `quasimont` the output files are overwritten.

3.3 STATIC-LIBRARY AND SOFTWARE INTEGRATION

Although we have demonstrated the effectiveness and consistency of QUASIMONT as stand-alone application, in this section we discuss its integration in larger mathematical software in numerical analysis and scientific simulations. We do this by providing **static library facilities** , to which the user can easily link to, when building any other C++ application. This option enables the user to integrate with ease the primary module `quasimont`, which is the one of interest from a computational perspective. First and foremost the static library `libquasimont.a` needs to be created by invoking the non-default rule `make static` of the library's source `makefile` in the aforementioned subdirectory.

Creation of libquasimont static library

```
user@machine: home/QUASIMONT> cd build/
user@machine: home/QUASIMONT/build> make static
g++ -c ../src/Quasimont.cpp -o obj/Quasimont.o -ansi -std=c++17 -I ../include
g++ -c ../src/DatIo.cpp -o obj/DatIo.o -ansi -std=c++17 -I ../include
g++ -c ../src/MonMap.cpp -o obj/MonMap.o -ansi -std=c++17 -I ../include
cd ../utilities && make
make[1]: Entering directory '/home/QUASIMONT/utilities'
g++ -c VecOps.cpp -o ../build/obj/VecOps.o -ansi -std=c++17 -I ../include
make[1]: Leaving directory '/home/QUASIMONT/utilities'
ar rcs ../static/libquasimont.a obj/Quasimont.o obj/DatIo.o obj/MonMap.o obj/VecOps.o
```

We chose to implement an example application using the static library of QUASIMONT in a separate directory **QUASIMONT/static**; by default the `make static` rule will create `libquasimont.a` in such directory. While building an external software, the user needs to specify absolute or relative paths (at her/his own discretion) to properly link the static library, i.e.:

- the **QUASIMONT/include** subdirectory containing the library header files, which is done by adding the flag `-Ipath/to/QUASIMONT/include` to the GCC compiler options. For our case this path variable will be `-I../include`;
- the actual `libquasimont.a` itself by adding `path/to/QUASIMONT/build/libquasimont.a` to the GCC linker option. For our case this path variable will be that of the current directory.

Therefore, if the user wants to build an external App using the static library QUASIMONT, those two GCC options must be appended to the commands specified for compiling and linking the user's own software. If the App is built using a makefile then those commands are to be added to the appropriate rules, as reported in the example below.

External software's makefile

```
default: App.o
  g++ App.o -o App libquasimont.a -lm -lquadmath -lgsl -lgslcblas

App.o: App.cpp
  g++  -c App.cpp -o App.o -ansi -std=c++17 -I../include
```

We recall that it is fundamental to correctly link third-party dependencies alongside QUASIMONT as reported at Section 2.1; failing to do so will result in undefined references. Moreover, the primary module's header file Quasimont.h needs to be included in the software's App.cpp source code in order to avoid undefined references to QUASIMONT's methods. This is clearly exemplified in the snippet below showing the content of the simplest external App using the primary method of our library.

App.cpp (user's external source code)

```cpp
#include "Quasimont.h"

int main()
{
  // TEST ON STATIC LIBRARY THIRD-PARTY INTEGRATION
  std::vector<float128> coeff_sequence = {PI, 3};
  std::vector<float128> muntz_sequence = {-1/E, 1.0/2.0};
  quasimont(muntz_sequence, coeff_sequence);

  std::ifstream nodes_txt, weights_txt;
  std::vector<float128> nodes, weights;
  float128 loaded_value;
  // LOAD-IN COMPUTED NODES
  nodes_txt.open("output/Nodes.txt");
  while(nodes_txt >> loaded_value)
  {
    nodes.push_back(loaded_value);
  }
  nodes_txt.close();
  // LOAD-IN COMPUTED WEIGHTS
  weights_txt.open("output/Weights.txt");
  while(weights_txt >> loaded_value)
  {
    weights.push_back(loaded_value);
  }
  weights_txt.close();

  // COMPUTE QUADRATURE LOADED NODES AND WEIGHTS
  float128 In = 0;
  for(int k=0; k < muntz_sequence.size(); k++)
  {
    float128 In_mon = 0;
    for(int j=0; j<nodes.size(); j++)
    {
      In_mon += weights[j]*pow(static_cast<float128>(nodes[j]),muntz_sequence[k]);
    }
    In += coeff_sequence[k]*In_mon;
  }
  // PRINT COMPUTED QUADRATURE
  std::cout << "\n\n ** I_n(p(x)) = " << In << "  [with reloaded parameters in double
    ↪ precision] **" << std::endl;
  return 0;
}
```

In this App the user instantiates the access point quasimont for an input polynomial $p(x) = \pi x^{-\frac{1}{e}} + 3x^{\frac{1}{2}}$ by specifying its coefficients and exponents sequence. QUASIMONT will then execute and outputting the results in the **output** subdirectory. From it the user can easily retrieve the new, optimised monomial transformation quadrature rule's parameters and re-load them back in App ; those new monomial transformed quadrature samples and weights can be used for whatever purpose the user intends to e.g. computing the numerical integrals of specific generalised polynomials. As reported below, the integral of $p(x)$ computed by QUASIMONT and the one computed manually by re-loading the quadrature coincide, entailing no loss of information takes place while loading those from the text files discussed in the previous Section 3.2. This re-loading procedure allows the direct use of specific quadratures generated by QUASIMONT in any user's apps while retaining separated her/his code from the library itself; this feature allows the users of our library to keep the resulting executable as light and versatile as possible. The user should run the above script and verify that the program executes smoothly.

Execution of external source code using QUASIMONT' static library

```
Input polynomial p(x) =  +3.14159265*x^(-0.367879441) +3*x^(0.5)

** Accepted sequence of exponents **
   {-0.367879441, 0.5}
** Lambda_min = -0.367879441, Lambda_max = 0.5 **
--------------------------------------------------
** N_min = 14
** Beta_min = 7.32695513, Beta_max = 28.924995 **
** Transformation order = 16.5615229 **
--------------------------------------------------
** Using double f.p. format for nodes and weights **
** I(p(x))   = 6.9699264004508497 **
** I_n(p(x)) = 6.9699264004508496      [with parameters in float128 precision] **
** E_n(p(x)) = 1.2031606718315283e-17  [with parameters in float128 precision] **
** I_n(p(x)) = 6.9699264004508496      [with parameters in double precision] **
** E_n(p(x)) = 1.338886432916513e-17   [with parameters in double precision] **

** QUASIMONT HAS TERMINATED **

** I_n(p(x)) = 6.9699264004508496      [with reloaded parameters in double precision] **
```

The resulting $n = 14$ nodes and weights computed by QUASIMONT are reported Table 3.1 for reference. As a final remark we note that, because of the relative paths to the raw data (see Section 2.2), the folder **QUASIMONT/data** must necessarily be copied one level above the the software's App.cpp source code that instantiates quasimont from the static library.

Nodes and weights (double f.p. format) of the proposed quadrature		
$j \in \mathbb{N}$	$x_j \in (0, 1)$	$w_j \in \mathbb{R}^+$
1	1.4596052905532561e-36	6.1894206168766954e-35
2	1.1132574083627952e-24	2.0651035137532638e-23
3	2.4377731846127742e-18	2.8391986747117744e-17
4	4.4998382880885933e-14	3.7464568184030571e-13
5	6.4003132127985324e-11	4.057105095079503e-10
6	1.7781156719228688e-08	8.8748074290566141e-08
7	1.5559948578773301e-06	6.2192989916469207e-06
8	5.6557186314867493e-05	0.00018196919612818068
9	0.0010151600139893485	0.002615333542701164
10	0.010082613309455394	0.020446702552108884
11	0.059855653657067465	0.092358369558320871
12	0.22390628224095116	0.24661688102402748
13	0.54690717721044069	0.37649243764791818
14	0.89228314395297181	0.26128199802564411

Table 3.1: Nodes and weights obtained by a monomial transformation of order $r = 16.56$ applied to the G-L quadrature integrating $p(x)$ in (0,1) at double precision with $n = 14$ for this custom application.

If such subdirectory is not placed properly the library will fail to locate the necessary .csv files needed for its execution and subsequently throw an error at run-time. For our case no further action is necessary as the directory **QUASIMONT/data** is already placed one level.

3.4 SECONDARY MODULE: ADDITIONAL SUPPORTING AND VISUAL TOOLS

To conclude the user interface of the library, we illustrate the additional functionalities of QUASIMONT's secondary module. All the content of the secondary module is provided by the unique ErrTools.cpp source file where, aside from the main function, it contains four methods instantiated inside it. They provide:

- the exact same tabulated data reported in TabulatedErrorValues.csv of $\beta_{\min}(n)$ and $\beta_{\max}(n)$ that is available in **QUASIMONT/data** ;
- the plot of the novel asymptotic error estimate (1.8) for a given, user-specified, value of n, that is fundamental to design the monomial transformation quadrature rule.

The user can compile and execute the secondary module by invoking the rule tools from the application's top level makefile .

Compilation and execution of QUASIMONT's secondary module

```
user@machine: home/QUASIMONT/test> make tools
cd ../utilities && make tools
make[1]: Entering directory '/home/QUASIMONT/utilities'
g++ -o Tools ErrTools.cpp -lmpfr -I ../include
./Tools

Computing for n = 10
    beta_min = 13.5722147
    beta_max = 14.0059
Computing for n = 12
    beta_min = 8.54136147
    beta_max = 23.2002
Computing for n = ...
```

The results will be stored in the run-time created **QUASIMONT/utilities/estimate** subdirectory. If the users re-compiles the source code of the secondary-module by specifying a different value of n, the previously obtained plots and error values will not be over-written. The secondary module has two additional dependencies w.r.t. the primary module, which is part of the reason why we decided to keep them separated in their implementation. Those are

- **GNU MPFR** [2]: based on **GMP - GNU Multi Precision** library, and interfaced with QUASIMONT via Boost Multiprecision library's back-end wrapper classes, it provides access to *higher-than-quadruple* f.p. formats which are necessary for the evaluation of the Euler's Gamma function in the computation of the asymptotic error estimate (1.8).
- **Gnuplot** : the well-knonw cross-platform utility used for plotting scientific charts and used by the shell script plot.sh in the **QUASIMONT/utilities** subdirectory.

The above required third-party libraries are once again easily installed locally by exploiting the relative package managers of the Linux distro of reference

Installation of third-party libraries for the secondary module

```
user@machine: home> # For Debian-based distros (Ubuntu, Mint, Knoppix, Kali ...)
user@machine: home> sudo apt-get install libgmp3-dev
user@machine: home> sudo apt install libmpfr-dev
user@machine: home> # For RPM-based distros (CentOS, Fedora, SUSE, Scientific Linux ...)
user@machine: home> sudo yum install install gmp-devel mpfr
```

Chapter 4

MODULES DESCRIPTION

In this final chapter we provide the user with a reference description of all methods in the source code of the library in terms of implementation and behaviour. The aim is to allow the user to better understand the features of the library for an easy enhancement and/or integration in external software. The source code of each module is described in the comment block that proceeds the function definition in its source file, that is either DatIo.cpp, MonMap.cpp or VecOps.cpp (see Section 2.2). We recall that the declaration of each method can be found in the corresponding header file (located in the **QUASIMONT/include** subdirectory) where a comment above each method provides the *"coordinates"* for the user to retrieve the function description. We divide the chapter in three sections, each one of them dedicated to the description of the methods defined in one of the three source files. Each method is then described in the relative subsection, which takes its name. A full breakdown list of each method is reported here for reference

- `MonMap.cpp`
  - `computeLambdaMax;`
  - `computeNumNodes;`
  - `computeMapOrder;`
  - `computeParamsGl;`
  - `computeQuadGl;`
  - `computeExactError;`
- `DatIo.cpp`
  - `manageData;`
  - `streamMonMapData;`
  - `optimiseData;`
  - `exportNewData;`
- `VecOps.cpp`
  - `castVector;`
  - `doubleDotProduct;`

4.1 MONMAP.CPP

4.1.1 computeLambdaMax

MonMap.cpp/computeLambdaMax

```
/////////////////////////////////////////////////////////////////////////////////////
//
//       FUNCTION: lambda_max = computeLambdaMax(lambda_min, user_n)
//
//          INPUT: - lambda_min = minimum (and only) exponent in the input sequence
//                 - user_n = desired number of (quadrature) nodes provided by the user
//
//         OUTPUT: - lambda_max = additional exponent of the new binomial
//
//    DESCRIPTION: the monomial transformation quadrature rule processes the G-L nodes &
//                 weights for those polynomials characterised by an arbitrarly large
//                 gap between the terms of minimum and maximum degree. There might be
//                 cases however in which the user wants to integrate singular
//                 monomials; in those cases the library will require an additional,
//                 non-constant, term to be added to the monomial. It does so by either
//                 allowing the user to manually input the exponent of the additional
//                 term from the CLI or to specify the number of nodes to use in
//                 its application. In this last instance the following function
//                 automatically generates the resulting exponent of the additional
//                 term (lambda_max).
//
/////////////////////////////////////////////////////////////////////////////////////

float128 computeLambdaMax(float128& lambda_min, int num_nodes)
```

4.1.2 computeNumNodes

MonMap.cpp/computeNumNodes

```
/////////////////////////////////////////////////////////////////////////////////////
//
//       FUNCTION: n = computeNumNodes(lambda_min, lambda_max)
//
//          INPUT: - lambda_min = minimum exponent in the input "muntz_sequence"
//                                (strictly greater than -1)
//                 - lambda_max = maximum exponent in the input "muntz_sequence"
//
//         OUTPUT: - n = number of (quadrature) nodes computed as the only real solution
//                       of equation 1.18 reported in the doc/UserManual.pdf
//
//    DESCRIPTION: once the exponents in the terms with minimum and maximum degree in
//                 the user-input polynomial have been determined, this method
//                 solves equation 1.18, which is a 7-th degree polynomial in
//                 n derived by a linear regression of n in beta_min/beta_max.
//                 It then extracts the only real root whose integer floor will then be
//                 the minimum possible number of (quadrature) nodes to be used in the
//                 new quadrature formula to achieve double precision (the solver itself
//                 is a class' method implemented in GSL-GNU Scientific Library).
//
/////////////////////////////////////////////////////////////////////////////////////

int computeNumNodes(const float128& lambda_min, const float128& lambda_max)
```

4.1.3 computeMapOrder

MonMap.cpp/computeMapOrder

```
////////////////////////////////////////////////////////////////////////////////////////
//
//        FUNCTION: r = computeMapOrder({lambda_min, lambda_max}, {beta_min, beta_max})
//
//           INPUT: - {lambda_min, lambda_max} = output of function 'manageData'
//                  - {beta_min, beta_max} = output of function 'streamMonMapData'
//
//          OUTPUT: - r = value of the transformation order of the monomial map (gamma)
//
//     DESCRIPTION: the order of the monomial transformation that characterises the new
//                  samples of the quadrature rule obtained starting from those of the
//                  G-L quadrature formula is computed as a linear interpolation
//                  beetween the minimum and maximum bound outlined in the inequality
//                  1.17 of the doc/UserManual.pdf.
//
////////////////////////////////////////////////////////////////////////////////////////

double computeMapOrder(const std::vector<float128>& lambdas, const std::vector<float128>&
    ↪ betas)
```

4.1.4 computeParamsGl

MonMap.cpp/computeParamsGl

```
////////////////////////////////////////////////////////////////////////////////////////
//
//        FUNCTION: [{new_x, new_w, old_x, old_w}] = computeParamsGl(r, n_min)
//
//           INPUT: - r = output of function 'computeMapOrder'
//                  - n_min = output of function 'streamMonMapData'
//
//          OUTPUT: - {new_x, new_w} = new set of G-L quadrature nodes (x) and
//                                     weights (w) following the monomial map
//                  - {old_x, old_w} = classical set of G-L quadrature nodes (x) and
//                                     weights (w) following the affine map
//
//     DESCRIPTION: once the transformation order is available, the monomial map itself
//                  is applied to the G-L nodes and weights that have been previously
//                  mapped from [-1,1] to [0,1] via an affine (linear) map. The
//                  complete set of new and old nodes and weights (referred to as
//                  quadrature_parameters) are collected in a tuple and outputted by
//                  this method.
//
////////////////////////////////////////////////////////////////////////////////////////

std::tuple<std::vector<float128>, std::vector<float128>, std::vector<float128>, std::vector
    ↪ <float128>> computeParamsGl(const double& r, const int& n_min)
```

4.1.5 computeQuadGl

MonMap.cpp/computeQuadGl

```
//////////////////////////////////////////////////////////////////////////////////
//
//       FUNCTION: In = computeQuadGl(x, w, muntz_sequence, coeff_sequence)
//
//          INPUT: - x = output of function 'computeParamsGl' optimised by 'optimiseData'
//                 - w = output of function 'computeParamsGl' optimised by 'optimiseData'
//                 - muntz_sequence = sequence of real exponents of the polynomial
//                 - coeff_sequence = sequence of real coefficients of the polynomial
//
//         OUTPUT: - In = value of the numerical approximated integral for the user input
//
//    DESCRIPTION: every interpolatory quadrature rule approximates the definite integral
//                 by means of a weighted sum of the kernel's values on specific points
//                 along the integration interval (i.e. nodes); an interpolatory
//                 Gaussian quadrature formula is a quadrature rule whose nodes
//                 corresponds to the roots of a polynomial that is orthogonal in the
//                 integration interval to the weight function of the kernel. A G-L
//                 quadrature formula with n+1 nodes is a Gaussian formula for which the
//                 nodes corresponds to the roots of the Legendre n-th degree polynomial
//                 that is orthogonal to the weight function w(x) = 1 in [-1, 1]. This
//                 routine implements the computation of such weighted sum in a general
//                 fashion, i.e. regardless of the quadrature rules.
//
//////////////////////////////////////////////////////////////////////////////////

template<typename type1, typename type2>
float128 computeQuadGl(const std::vector<type1>& nodes,const std::vector<type1>& weights,
    ↪ std::vector<type2>& muntz_sequence, std::vector<type2>& coeff_sequence)
```

4.1.6 computeExactError

MonMap.cpp/computeExactError

```
//////////////////////////////////////////////////////////////////////////////////
//
//       FUNCTION: En = computeExactError({post_map_quadrature, pre_map_quadrature},
//                                        muntz_sequence, coeff_sequence, print_flag)
//
//          INPUT: - {post_map_quadrature, pre_map_quadrature} = output of function
//                                                              'computeQuadGl'
//                 - muntz_sequence = sequence of real exponents of the polynomial
//                 - coeff_sequence = sequence of real coefficients of the polynomial
//                 - print_flag = boolean parameter to tell the routine wheter or not
//                                to print the numerical value of the primitive
//
//         OUTPUT: - En = relative error of the specified rule computed with the new
//                        nodes and weights
//
//    DESCRIPTION: let I_n be the numerical integral calculated using the G-L quadrature
//                 rule (outputed by function 'computeQuadGl') and I_ex be the exact
//                 (analytic) value of such integral then the relative error of the
//                 quadrature can be computed a-posteriori as R_n = |I_ex - I_n|/|I_ex|.
//                 Such computation is implemented in this routine with the exact
//                 integral being as precise as the user-input polynomial is.
//
//////////////////////////////////////////////////////////////////////////////////

template<typename type>
float128 computeExactError(const float128& In, std::vector<type>& muntz_sequence, std::
    ↪ vector<type>& coeff_sequence, bool& print_primitive)
```

4.2 DATIO.CPP

4.2.1 manageData

DatIo.cpp/manageData

```
////////////////////////////////////////////////////////////////////////////////
//
//       FUNCTION: [n, {lambda_min, lambda_max}] = manageData(muntz_sequence,
//                                                            coeff_sequence)
//
//          INPUT: - muntz_sequence = sequence of real exponents of the polynomial
//                 - coeff_sequence = sequence of real coefficients of the polynomial
//
//         OUTPUT: - n = output of function 'computeNumNodes' or input by the user
//                 - lambda_min = minimum exponent in the input "muntz_sequence"
//                 - lambda_max = maximum exponent in the input "muntz_sequence"
//
//    DESCRIPTION: the user-input polynomial is provided to the library via Main.cpp; the
//                 polynomial itself is specified via a unsorted sequence of
//                 coefficients and exponents of the various monomials in the polynomial.
//                 Once those input are read, checks have to be made in order to validate
//                 the proper functioning of the library; those are:
//                    - the number of exponents and the number of coefficients coincide;
//                    - the input polynomial is at least a binomial (otherwise the
//                      further CLI user-input is required, see Section 2.4 in the
//                      doc/UserManual.pdf;
//                    - lambda_min > -1 (otherwise the input sequence of exponents is
//                      not a Muntz sequence and thus the program exits);
//                 Once those checks are ran the exponents' sequence is sorted locally
//                 and lambda_min/lambda_max are thus identified and outputted alongside
//                 the associated number of nodes computed by the function
//                 'computeNumNodes' (see lines 57~121 in the 'src/MonMap.cpp' file).
//
////////////////////////////////////////////////////////////////////////////////

template<typename type>
std::tuple<int, std::vector<float128>> manageData(std::vector<type>& muntz_sequence, std::
    ↪ vector<type>& coeff_sequence)
```

4.2.2 streamMonMapData

DatIo.cpp/streamMonMapData

```
////////////////////////////////////////////////////////////////////////////////
//
//       FUNCTION: [n_min, {beta_min, beta_max}] = streamMonMapData(n)
//
//          INPUT: - n = output of function 'manageData'
//
//         OUTPUT: - n_min = minimum possible number of nodes listed in the
//                           'data/TabulatedErrorValues.csv' file to obtain
//                           double-precise numerical integration
//                 - beta_min = minimum value for the exponent of the post-map polynomial
//                 - beta_max = maximum value for the exponent of the post-map polynomial
//
//    DESCRIPTION: the monomial transformation gamma: [0,1] -> [0,1] is uniquely
//                 identified by its order r which in turn requires the knowledge of
//                 beta_min/beta_max. This method scans the tabulated vales in the
//                 'data/TabulatedErrorValues.csv' file to extract such values
//                 according to the input n (number of quadrature samples).
//
////////////////////////////////////////////////////////////////////////////////

std::tuple<int, std::vector<float128>> streamMonMapData(const int& comp_num_nodes)
```

4.2.3 optimiseData

DatIo.cpp/optimiseData

```
/////////////////////////////////////////////////////////////////////////////////////////
//
//       FUNCTION: optimiseData(quadrature_parameters, muntz_sequence, coeff_sequence)
//
//          INPUT: - quadrature_parameters = output of function 'computeParamsGl'
//                 - muntz_sequence = sequence of real exponents of the polynomial
//                 - coeff_sequence = sequence of real coefficients of the polynomial
//
//         OUTPUT: no outputs
//
//    DESCRIPTION: the declared objective of the library is to exploit the defining
//                 characteristic of the input singular polynomial to derive an ad-hoc
//                 quadrature scheme whose new nodes and weights are redistributed to
//                 better capture the singularity while retaining the lowest possible
//                 computational complexity. Such measure (threshold of accuracy for E_n)
//                 is set to be the machine-epsilon in double precision by default,
//                 although more and less stringent constraints can be inserted. Once the
//                 new set of nodes and weights have been computed, this function
//                 optimises their floating-point representation.
//
/////////////////////////////////////////////////////////////////////////////////////////

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)
```

4.2.4 exportNewData

DatIo.cpp/exportNewData

```
/////////////////////////////////////////////////////////////////////////////////////////
//
//       FUNCTION: exportNewData(optim_nodes, optim_weights, [I_new, E_new, I_old, E_old])
//
//          INPUT: - optim_nodes = output of function 'computeParamsGl' optimised by
//                                'optimiseData'
//                 - optim_weights = output of function 'computeParamsGl' optimised by
//                                'optimiseData'
//                 - [I_new, E_new, I_old, E_old] = values of the numerical quadrature
//                                                  (I_new, I_old) and its relative
//                                                  error (E_new, E_old) obtained with
//                                                  the new samples provided by the
//                                                  monomial transformation quadrature
//                                                  rule and by the classical G-L
//                                                  samples
//         OUTPUT: no outputs
//
//    DESCRIPTION: once the monomial transformation quadrature rule is completed data is
//                 streamed in output text files located in the application's dir.
//                 The output data is organised in three separate files:
//                    - 'Results.txt' containins recap informations about the monomial
//                      transformation quadrature rule such as the parameters of the map
//                      (gamma), number of quadrature samples (n_min), etc...
//                    - 'Nodes.txt' listing the new nodes computed by the quadrature rule
//                    - 'Weights.txt' listing the new weights of the above rule
//
/////////////////////////////////////////////////////////////////////////////////////////

template<typename type>
void exportNewData(const std::vector<type>& nodes, const std::vector<type>& weights, const
    std::vector<float128>& output_data)
```

4.3 VECOPS.CPP

4.3.1 castVector

VecOps.cpp/castVector

```
/////////////////////////////////////////////////////////////////////////////////////////
//
//       FUNCTION: output_vector = castVector(input_vector, type_output)
//
//          INPUT: - input_vector = vector of length n of type of type T1
//                 - type_output = floating-point data-type T2
//
//         OUTPUT: - output_vector = input_vector casted in type T2
//
//    DESCRIPTION: this method casts the T1=float128 input vector to the an output vector
//                 with the same content but type T2 specified by the user.
//
/////////////////////////////////////////////////////////////////////////////////////////

template<typename type>
std::vector<type> castVector(const std::vector<float128>& input_vector, const type&
    ↪ type_infer)
```

4.3.2 doubleDotProduct

Utils.cpp/doubleDotProduct

```
/////////////////////////////////////////////////////////////////////////////////////////
//
//       FUNCTION: inner_product = doubleDotproduct(x, w)
//
//          INPUT: - x = vector containing the quadrature nodes in format float128
//                 - w = vector containing the quadrature weights in either float128 or
//                       double
//
//         OUTPUT: - inner_product = precise inner product between x and w avoiding
//                                   numerical cancellation
//
//    DESCRIPTION: This method takes two input vectors of the same length n, sorts
//                 their element-wise multiplication in a new vector with cells
//                 arranged in ascending order and sums them thereby assuring that no
//                 numerical cancellation occurs.
//
/////////////////////////////////////////////////////////////////////////////////////////

template<typename type>
float128 doubleDotProduct(const std::vector<float128>& f_values, const std::vector<type>&
    ↪ weights)
```

4.4 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] *Boost C++ Libraries.* URL: https://www.boost.org/.

[2] Laurent Fousse et al. "MPFR: A Multiple-Precision Binary Floating-Point Library with Correct Rounding". In: *ACM Trans. Math. Softw.* 33.2 (2007), 13–es. ISSN: 0098-3500. DOI: 10.1145/1236463.1236468.

[3] Walter 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.

[4] *GNU Compiler Collection.* URL: https://gcc.gnu.org/.

[5] *GNU Make.* URL: https://www.gnu.org/software/make/.

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

[7] Roberto 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] Roberto D. Graglia and Guido 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] Nicholas Hale and Alex 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] Guido 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: https://doi.org/10.1002/nme.2684.

[12] Guido Lombardi and Davide Papapicco. "A C++ library for high precision Gaussian quadrature of generalised/singular polynomials of non-integer degree". In: *ACM Trans. Math. Softw.* ?? (?), ? DOI: ?.

[13] Jin D. Ma, Vladimir Rokhlin, and Stephen 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] Gradimir V. Milovanović, Tomislav S. Igić, and Dragana 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: https://doi.org/10.1016/j.cam.2014.10.009.