# 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

**1 Preliminaries** **1**

1.1 Overview . . . 1

1.2 Mathematical background . . . 3

1.2.1 Interpolatory quadrature . . . 3

1.2.2 Gaussian quadrature formulae . . . 3

1.2.3 Asymptotic error estimation . . . 4

1.2.4 Müntz theorem and generalised polynomials . . . 4

1.2.5 Integration interval . . . 5

1.2.6 Monomial transformation . . . 6

**2 Installation** **9**

2.1 Third-party code . . . 9

2.2 Structure . . . 13

2.3 Build process . . . 16

2.4 Installation and test driver . . . 18

**3 User interface** **23**

3.1 The input source file . . . 23

3.2 Results and outputs . . . 24

3.3 Static-library and software integration . . . 26

3.4 Modular calls for independent integration/development . . . 29

3.5 Additional supporting and visual tools . . . 31

**4 Modules description** **33**

4.1 MonMap.cpp . . . 34

4.1.1 computeLambdaMax . . . 34

4.1.2 computeNumNodes . . . 34

4.1.3 computeMapOrder . . . 35

4.1.4 computeQuadParams . . . 35

4.1.5 computeQuadRule . . . 36

4.1.6 computeExactError . . . 36

4.2 DatIo.cpp . . . 37

4.2.1 manageData . . . 37

4.2.2 streamMonMapData . . . 37

4.2.3 optimiseData . . . 38

4.2.4 exportNewData . . . 38

4.3 VecOps.cpp . . . 39

4.3.1 castVector . . . 39

4.3.2 doubleDotProduct . . . 39

4.4 Acknowledgements . . . 39

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 its own package generating specialised quadrature rules for sets of generalised polynomials or by integrating it in external custom application. The principles behind the implementation of the library are discussed in this first chapter, followed by instructions for correctly building and running the library in a Linux distribution (outlined in the second and third chapter) and finally by a concluding chapter that briefly describes each of its modules. QUASIMONT offers its users a high degree of flexibility concerning its usage and interaction, specifically:

- it can be compiled and executed either as a stand-alone software implementing high-precision quadrature rules (see Section 2.4 and 3.1) or by instantiating its functionalities in external code (see Section 3.2);
- it provides two *modes* of execution (namely the *loud* and *silent* modes) by limiting the terminal's output information and streaming out the results of its execution (see Section 3.2);
- it can be integrated as an accessory tool in any C++ third-party application by linking the compiled code to its static library (see Section 3.3);
- it can be deployed as a supporting utility to generate, export and manipulate the new optimised samples and weights for a given generalised polynomial (see Section 3.3 and 3.4).

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 the monomial transformation, constituting the innovative core of the library, 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 pivot algorithm around which the library has been built (i.e. the monomial transformation) has its roots in the relative error estimation of the Gauss-Legendre (G-L) quadrature rule and on the ad-hoc numerical manipulation of the integration parameters (discussed in the following sections) according to the characteristics of the 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 work in [12] reports the automatised implementation and performance of the library. The monomial transformation quadrature rule is a result of the application of a novel exact asymptotic error estimate of the G-L quadrature rules introduced in [11] that shows the potentialities of such formulae to integrate generalized polynomials composed by monomials of 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 shown in the test drivers (see Section 2.4), QUASIMONT significantly outperforms the classical G-L quadrature rule for polynomials whose exponents' sequence is composed of either rational or irrational numbers (henceforth referred to as **Müntz sequence** , see Sub-Section 1.2.4). To consistently achieve such performance, our package implements a routine based on the following fundamental steps (discussed in more detail in the second chapter of this manual):

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

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 do suffice). Indeed, the core aim of the library is the efficient and precise integration of generalised polynomials where other techniques may require heavy computational cost at run-time [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 below), provided in the **Boost's Multiprecision** library, without which the precise computation of the quadrature parameters and the quadrature itself, for the most computationally demanding cases, would have not been possible. QUASIMONT implements methods that suit a very specific demand in numerical analysis so we used a procedural/modular approach in its writing (as opposed to an object-oriented paradigm) in which all the different functions interact through the quasimont module (see Section 2.1 and 2.2).

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

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

## 1.2 MATHEMATICAL BACKGROUND

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

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

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

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

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

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

where the factorised $g(x)$ always contains its most regular part whereas $w(x)$, known as the **weight function** contains, if present, any irregular and/or singular part of $f(x)$. If the weight function is not null everywhere in $(a, b)$ and we can find an infinite sequence of classical polynomials $P := \{p_j(x) \in \mathbb{P}_j\}_{j=0,1,\dots}$ 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 $\delta_{jk}$ 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, throughout the years they have been generalised using refined approximations and numerical efforts [3]. QUASIMONT integrates generalised polynomials which are extensively used in modelling several cases of particular relevance and interest in mathematical modelling and scientific computing, as mentioned above. The simplest and arguably most widely used Gaussian quadrature rule is the Gauss-Legendre (G-L) formula, for which the sequence $P$ is made of **Legendre's polynomials** that are orthogonal w.r.t. the constant weight function $w(x) = 1$ over the interval $(a = -1\,,\ b = +1)$.

## 1.2.3 ASYMPTOTIC ERROR ESTIMATION

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

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

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

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

where $E_n(f)$ is the estimated reminder of the quadrature rule. In [11], one of the authors of the present work, derived a closed form a-priori error asymptotic estimation of $E_n(f)$ for a G-L formula applied to a 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 polynomial integrand function (see Sub-Section 3.3 in [12] and references therein). In particular, given a specified number of quadrature nodes $n$, we can clearly see in Figure 1.1 how the a-priori relative error estimate $R_n(x^\lambda)$ behaves asymptotically with $\lambda$. For instance we immediately notice that the value of $R_n(f)$, with a fixed $n$ samples, is below a pre-determined arbitrary precision (in terms of relative error) within a specific, finite range of values of $\lambda$ 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.

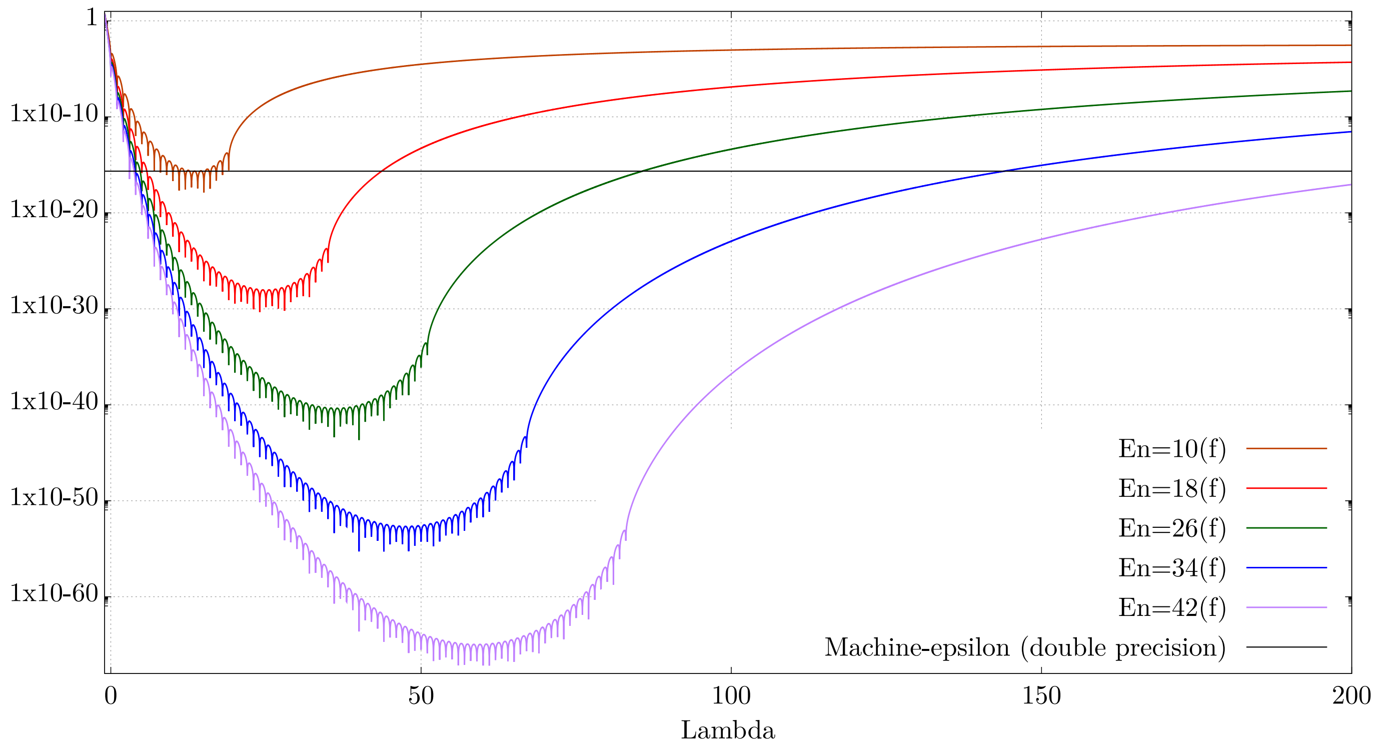


Figure 1.1: Evaluation of the relative error estimate $R_n(x^\lambda)$ provided in (1.8) for $n = 10, 18, 26, 34, 42$. We note that for a fixed $n$ we have that the relative error is below an arbitrary (pre-determined) precision threshold in $\forall\lambda \in (\lambda_{min}, \lambda_{max})$. In the figure the solid horizontal line shows f.p. double precision. We also observe that as long as $\lambda$ 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 $\lambda$. This plot has been realised using the secondary module of QUASIMONT (see Section 3.5).

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 $\Lambda$ 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 $\lambda_r$ defined in $(0, 1]$ where $\Lambda_r := \{\lambda_r > \cdots > \lambda_j > \lambda_{j-1} > \cdots > \lambda_1 > -1\}$, $r \in \mathbb{N}_0$. In the following we shall assume that any polynomial with $r$ terms having at least one constituting monomial of non-integer degree are to be considered Müntz (generalised) polynomials. 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 the G-L quadrature's performance (see in Figure 1.1 the relative error falls below the machine-epsilon threshold in double f.p. format) with an ad-hoc designed monomial transformation.

### 1.2.5 INTEGRATION INTERVAL

As previously stated and also reported in [12], 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 consistent 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 may be affected by numerical cancellation. Often, in practical problems, integration on an arbitrary interval $(a, b)$ is requested. For this reason, we need to define an affine transformation to map the original, generic, integration interval $(a, b)$ into $(0, 1)$. To that purpose, let us introduce a (linear) affine map $\varphi : (a, b) \to (0, 1)$ specified as

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

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

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

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

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

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. The Table 1.2 below lists the classical G-L nodes and weights alongside their (1.16) and (1.19) respective counterparts used by the proposed monomial transformation quadrature rule.

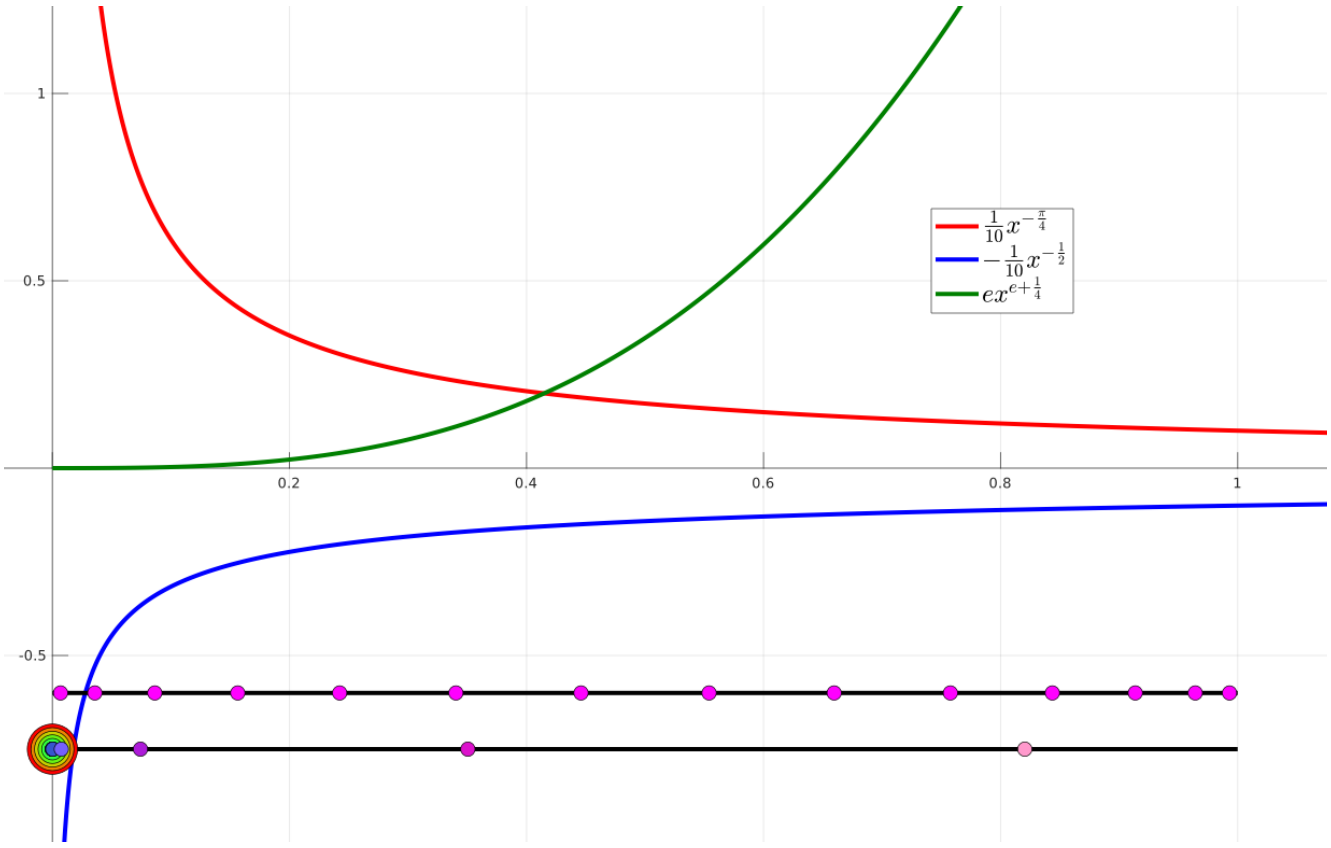


Figure 1.2: Plot of three generalised monomial terms of the Müntz polynomial $p(x)$ in (1.20). On the bottom of the graph there is the distribution in $(0, 1)$ of the $n = 14$ nodes of a classical G-L formula (magenta) and the new samples obtained by the monomial transformation quadrature rule (linear 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. $x = 0$.

| | **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 $r = 28.77$ as depicted by Figure 1.2.

Chapter 2

# INSTALLATION

In the following chapter we illustrate how to build executable applications with QUASIMONT from its source code as well as the dependencies needed at link-time. 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. We remark that all of the following content regards the execution of QUASIMONT as a stand-alone software as we postpone to the next chapter the instructions needed to use our tool as a static library that is fully-integrable into a user's own application. Finally we address what has been originally mentioned in the opening lines of the first chapter of the present manual, that is that our package features two *modes* of execution, namely the **loud** and **silent mode** . Although the content of the present chapter is independent of the *mode* with which QUASIMONT is executed, we hereby refer exclusively to the former loud mode while for the latter we direct the user to Section 3.2 in which we address the main differences in I/O between the two.

## 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 3.5 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; those are listed below with their minimum versions required for the correct build and execution of QUASIMONT reported between brackets:

- **Boost C++ libraries (v-1.66)** [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` 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 standard search paths in order to correctly link the objects files compiled from the source code. All of the listed requirements are easily installed in Linux using package managers, such as the **Advanced Package Tool** and the **Yellowdog Updater, Modified** (or its successor **Dandified YUM**), with a few commands on the terminal.

Installation of third-party libraries

```
user@machine: home> # For Debian-based distros (Ubuntu, Mint, Knoppix, Kali ...)
user@machine: home> sudo apt-get update
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 dnf makecache --refresh
user@machine: home> sudo dnf install http://repo.okay.com.mx/centos/8/x86_64/release/okay-
    ↪ release-1-1.noarch.rpm
user@machine: home> sudo dnf install boost gsl libquadmath
user@machine: home> sudo dnf 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 building process over cross-platform compliance by adopting the usage of *makefiles* (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 dnf 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 the minimum versions `8.5.0` and `4.2.1` of `gcc` and `make` respectively. We then tested its compilation and execution on three different platforms, namely using two Intel(R) Core(TM) i5-1035G1 @ 1.00GHz and i9-9900K @ 3.60GHz processors on a DEB-based `Ubuntu 20.04.3` operating system (OS) and also a Intel(R) Core(TM) i7-10700 @ 2.90GHz processor on a RPM-based `CentOS 8.5` OS (the last two running on virtual machines on a Windows host). Although the library has been tested in multiple development configurations that are compatible with the above listed requirements, the user may refer to Table 2.1 where we recap the information regarding the aforementioned authors' specific development environments. Once all the above packages have been installed, the user should thoroughly check the correct configuration, having care to particularly assure that links to the static libraries of each of the dependencies listed above have been added to `gcc`'s linker `ld` standard search paths.

| **Environments recap** | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Env. n. | OS (Unix) | Processor | GCC | Make | Boost | GSL | MPFR | GMP |
| 1 | Ubuntu 20.04.3 | i5-1035G1 | 9.3.0 | 4.2.1 | 1.77 | 2.5 | 4.1.0 | 6.2.1 |
| 2 | CentOS 8.5 | i7-10700 | 8.5.0 | 4.2.1 | 1.66 | 2.5 | 4.1.6 | 5.1.0 |

Table 2.1: QUASIMONT Development environments used by the authors. The two rightmost columns refer to the secondary module of the library and are not a dependency requirement (see Section 3.5).

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.

Quasimont.h

```
//---------------------------------------------------------------------------------
// File:       include/Quasimont.h
//
// Library:    QUASIMONT-QUAdrature of SIngular polynomials using MONomial Transformations:
//                      a C++ library for high precision integration of generalised
//                      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

// Loud mode
template<typename T>
void quasimont(std::vector<T>& muntz_sequence, std::vector<T>& coeff_sequence);
// Silent mode
std::vector<std::vector<double>> quasimont(double lambda_min, double lambda_max);

#endif // QUASIMONT_H
```

As for any other source file in the library, a comment block precedes the code; in it the first line specifies the location of the specific file in the library's relative directory's tree (see Figure 2.1). We only report such comment box for the above snippet of Quasimont.h and omit it for any following snippet for the sake of space and clarity.

Quasimont.cpp

```cpp
#include "Quasimont.h"

// Global variable controlling the primary module mode of execution
bool loud_mode = true;

// LOUD MODE
template<typename T>
void quasimont(std::vector<T>& muntz_sequence, std::vector<T>& coeff_sequence)
{
  // Print initial message and selects user's inputs
  auto input_data = manageData(muntz_sequence, coeff_sequence);

  // Extract beta_min, beta_max and n_min
  auto monomial_data = streamMonMapData(std::get<0>(input_data));

  // Compute order of the monomial transformation
  double transf_order = computeMapOrder(std::get<1>(input_data), std::get<1>(monomial_data)
    ↪ );

  // Compute the new nodes and weights of the Monomial Transformation Quadrature Rule
  auto quad_data = computeQuadParams(transf_order, std::get<0>(monomial_data));

  // Cast the quadrature parameter in the most optimised f.p. format possible
  optimiseData(quad_data, muntz_sequence, coeff_sequence);
}
template void quasimont<float128>(std::vector<float128>& muntz_sequence, std::vector<
    ↪ float128>& coeff_sequence);
template void quasimont<double>(std::vector<double>& muntz_sequence, std::vector<double>&
    ↪ coeff_sequence);

// SILENT MODE
std::vector<std::vector<double>> quasimont(double lambda_min, double lambda_max)
{
  // Deactivate terminal's and files' output
  loud_mode = false;

  // Initialise input parameters of the Monomial transformation quadrature rule
  std::vector<double> muntz_sequence = {lambda_min, lambda_max};
  std::vector<double> coeff_sequence = {1.0, 1.0};

  // Print initial message and selects user's inputs
  auto input_data = manageData(muntz_sequence, coeff_sequence);

  // Extract beta_min, beta_max and n_min
  auto monomial_data = streamMonMapData(std::get<0>(input_data));

  // Compute order of the monomial transformation
  double transf_order = computeMapOrder(std::get<1>(input_data), std::get<1>(monomial_data)
    ↪ );

  // Compute the new nodes and weights of the Monomial Transformation Quadrature Rule
  auto quad_data = computeQuadParams(transf_order, std::get<0>(monomial_data));

  // Cast the quadrature parameter in the most optimised f.p. format possible
  optimiseData(quad_data, muntz_sequence, coeff_sequence);

  // Generate double-precise new nodes and weights and export them in memory as output
  std::vector<double> nodes = castVector(std::get<0>(quad_data), std::numeric_limits<double
    ↪ >::epsilon());
  std::vector<double> weights = castVector(std::get<1>(quad_data), std::numeric_limits<
    ↪ double>::epsilon());
  return std::vector<std::vector<double>> {nodes, weights};
}
```

We note that all the headers (and thus the declarations of all their methods) constituting the library (see Section 2.2) are included in Quasimont.h . Further below in the above header we find the definitions of the important constants that are used throughout the library which precede 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 user may notices that we overloaded the primary module quasimont method by taking different inputs and returning different outputs depending on whether the loud or silent mode is executed. We observe that compile-time polymorphism in C++ is not allowed when only the return type of the function differs which results in identical signatures. The definition for both modes of the method is given in the homonym source file Quasimont.cpp reported above.

## 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. All of QUASIMONT's 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
#ifndef MONMAP_H
#define MONMAP_H

// (SEE LINES 18~37 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 53~70 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 112~125 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 184~202 IN 'src/MonMap.cpp') Computes the new nodes and weights of the monomial transformation quadrature rule
std::tuple<std::vector<float128>, std::vector<float128>, std::vector<float128>, std::vector<float128>> computeQuadParams(const double& r, const int& n_min);
// (SEE LINES 309~328 IN 'src/MonMap.cpp') Computes the numerical integral for a given quadrature rule
template<typename type1, typename type2>
float128 computeQuadRule(const std::vector<type1>& nodes, const std::vector<type1>& weights, std::vector<type2>& muntz_sequence, std::vector<type2>& coeff_sequence);
// (SEE LINES 356~378 IN 'src/MonMap.cpp') Computes the a-posteriori relative error of the quadrature rule
template<typename type>
float128 computeExactError(const float128& In, std::vector<type>& muntz_sequence, std::vector<type>& coeff_sequence, bool& print_primitive);

#endif // 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 header below

DatIo.h

```cpp
#ifndef DATIO_H
#define DATIO_H

// (SEE LINES 18~42 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 211~229 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 274~294 IN 'src/DatIo.cpp') Degrade the precision of the new nodes and
    ↪ weights to establish minimum data-type for double precision quadrature
template<typename type>
void optimiseData(std::tuple<std::vector<float128>, std::vector<float128>, std::vector<
    ↪ float128>, std::vector<float128>>& quad_params, std::vector<type>& muntz_sequence,
    ↪ std::vector<type>& coeff_sequence);
// (SEE LINES 392~417 IN 'src/DatIo.cpp') Computes and exports the transformed weights and
    ↪ nodes along with other ouputs
template<typename type>
void exportNewData(const std::vector<type>& nodes, const std::vector<type>& weights, const
    ↪ std::vector<float128>& output_data);

#endif // DATIO_H
```

- VecOps.cpp defines every function that neither performs quadrature-related computations nor I/O data operations. Given their generic nature, no naming scheme is assigned to them as they are declared in Utils.h alongside the 3 methods of the secondary module defined in ErrTools.cpp (see Section 3.5).

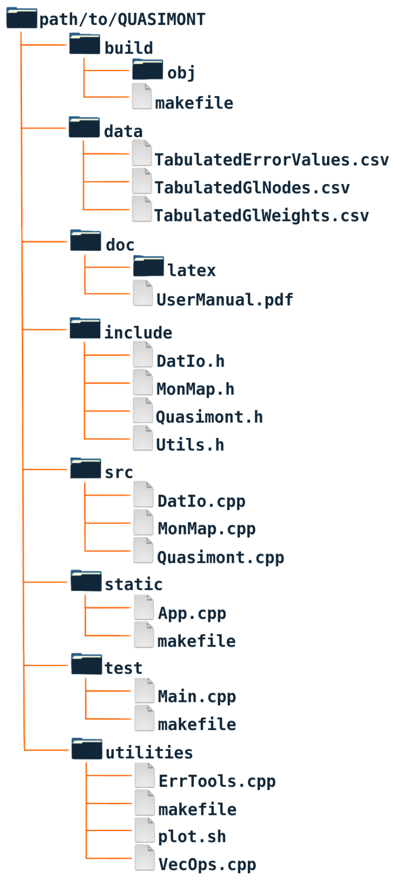
Utils.h

```cpp
#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 instantiation
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 supplementary **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.



```
path/to/QUASIMONT
├── build
│   ├── obj
│   └── makefile
├── data
│   ├── TabulatedErrorValues.csv
│   ├── TabulatedGlNodes.csv
│   └── TabulatedGlWeights.csv
├── doc
│   ├── latex
│   └── UserManual.pdf
├── include
│   ├── DatIo.h
│   ├── MonMap.h
│   ├── Quasimont.h
│   └── Utils.h
├── src
│   ├── DatIo.cpp
│   ├── MonMap.cpp
│   └── Quasimont.cpp
├── static
│   ├── App.cpp
│   └── makefile
├── test
│   ├── Main.cpp
│   └── makefile
└── utilities
    ├── ErrTools.cpp
    ├── makefile
    ├── plot.sh
    └── VecOps.cpp
```

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 Section 3.3).

## 2.3 BUILD PROCESS

QUASIMONT is a quadrature tool that can be easily integrated in users' applications, while creating executables. 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

```
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 #-fext-numeric-literals
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 deletes those object files and the source code does 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 sub-directory as described in 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. Regardless, during compilation it may happen that, depending on the distro and version of gcc used, the user is prompted with the following error: unable to find numeric literal operator 'operator""Q' . In such case the user must append the -fext-numeric-literals flag to the CXXOPTIONS macro as reported in the above makefile snippet and all the remaining ones in the library (see Section 2.2 and Figure 2.1). 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 are 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 for three generalised and classical polynomial, thus introducing the user to its execution and interaction 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 $x = 0$ (QUASIMONT defaults the integration interval to be $(0, 1)$). Once the library's and the input source file have been compiled and linked into the corresponding executable, the resulting application integrates each polynomial using the monomial transformation quadrature rule. We remark that, although no difference in numerical results occurs, the following report solely concerns the loud mode of execution. A list of information regarding the monomial transformation itself is displayed on the terminal whereas the results of interest (i.e. the new processed quadrature together with the related numerical integration) are exported in files located in the appropriate output directory (see Section 3.2).

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 **
```

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 2.2, 2.3, 2.4 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 $(0, 1)$. The Table 2.2 below shows what was graphically reported in Figure 1.2 i.e. the ad-hoc suited and more efficient distribution of the samples obtained by the monomial transformation as opposed to the ones of the classical G-L quadrature rule.

| **Nodes and weights of the proposed quadrature for $p_1(x)$** | | |
|---|---|---|
| $j \in \mathbb{N}$ | $x_j \in (0,1)$ | $w_j \in \mathbb{R}^+$ |
| 1 | 4.0256721894941735e-83 | 2.9709584266857193e-81 |
| 2 | 2.2116841854653406e-62 | 7.1971053989801097e-61 |
| 3 | 3.4370358897566318e-51 | 7.1255611223692978e-50 |
| 4 | 1.6010758830624544e-43 | 2.4253789034229506e-42 |
| 5 | 1.0479208102676717e-37 | 1.2456363438765983e-36 |
| 6 | 4.8718071131884711e-33 | 4.7453741025535338e-32 |
| 7 | 3.7119919463646725e-29 | 3.0494845207591382e-28 |
| 8 | 7.5510919371932376e-26 | 5.3393428605147779e-25 |
| 9 | 5.58947068987428e-23 | 3.4526789629984772e-22 |
| 10 | 1.8549840400142102e-20 | 1.0121884313459638e-19 |
| 11 | 3.1981531227726135e-18 | 1.5545919457374846e-17 |
| 12 | 3.1905126409913618e-16 | 1.3904418738462426e-15 |
| 13 | 1.9974633805093215e-14 | 7.8421013950383668e-14 |
| 14 | 8.3548669580378659e-13 | 2.9653252968844048e-12 |
| 15 | 2.4525558094710645e-11 | 7.8879379262599627e-11 |
| 16 | 5.2553535644885825e-10 | 1.5337625816699302e-09 |
| 17 | 8.4865615532959245e-09 | 2.2485149221054993e-08 |
| 18 | 1.0601166175497224e-07 | 2.5487504629916981e-07 |
| 19 | 1.0467771915132323e-06 | 2.2805293469267595e-06 |
| 20 | 8.3187859060800558e-06 | 1.6383778757563812e-05 |
| 21 | 5.4017656350748472e-05 | 9.583886949817516e-05 |
| 22 | 0.00029027719974030597 | 0.00046172220499588948 |
| 23 | 0.0013048618829373392 | 0.0018488842315729846 |
| 24 | 0.0049515587664587246 | 0.0061972140073356637 |
| 25 | 0.01598386920155067 | 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 2.2: Nodes and weights obtained by a monomial transformation of order $r = 28.77$ applied to the G-L quadrature integrating $p_1(x)$ in $(0, 1)$ at double precision with $n = 32$ samples.

While in fact we know that the $n = 32$ nodes of the classical G-L quadrature rule are distributed symmetrically in $(0, 1)$ with an equal abundance of those at both bounds of the integration interval, our proposed quadrature scheme, based on the monomial transformation, concentrates the samples close to the lower bound $x = 0$, squashing the vast majority of them (26 out of the total 32) inside the sub-interval $(0, 0.1)$ thereby better capturing the singular behaviour of $p_1(x)$. 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 allows to integrate such integrals identifying the degree of the monomial as $\lambda_{\min}$ and the maximum degree $\lambda_{\max}$ of the all set of combined polynomial functions. For this reason, in this test, we add a monomial with $\lambda_{\max}$ degree and unitary coefficient to $p_2(x)$. Once constructed the final polynomial we apply the proposed algorithm to process the optimised nodes and weights. The integration is enabled using a caveat prompted on the terminal and it consists in choosing one path between two mutually exclusive options that the user must exercise in order to continue with the application.

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:
```

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 $\tilde{p_2}(x) = x^{-\frac{e}{3}} + x^{\lambda_{\max}}$ 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 causes the application to resort to its original workflow with the additional intermediate step of requiring the input value of $\lambda_{\max}$. We remark that any miss-typed or empty input results in QUASIMONT to throw an error message and exit the program.

Building and executing the test driver: p_2(x)

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

 ** QUASIMONT HAS TERMINATED **
```

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 observe that the latter format has been selected despite being slightly less precise than the former. The logic behind is that both relative errors are computed at run-time and if their absolute difference is smaller than twice the machine-epsilon in double precision than we go ahead and export the data in the most optimised format.

| **Nodes and weights of the proposed quadrature for $p_2(x)$** | | | | | |
|---|---|---|---|---|---|
| $j \in \mathbb{N}$ | $x_j \in (0,1)$ | $w_j \in \mathbb{R}^+$ | $j \in \mathbb{N}$ | $x_j \in (0,1)$ | $w_j \in \mathbb{R}^+$ |
| 1 | 2.7630147554775799e-157 | 3.2089404684705014e-155 | 33 | 7.0500634584281311e-14 | 1.5167370865604578e-13 |
| 2 | 1.2504818169246583e-124 | 6.4130575288015817e-123 | 34 | 5.7559583188242479e-13 | 1.1793837946815465e-12 |
| 3 | 5.6814585252048909e-107 | 1.8611939310882479e-105 | 35 | 4.2517694749483687e-12 | 8.2956748948158666e-12 |
| 4 | 7.80153953804910668e-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 2.3: Nodes and weights obtained by a monomial transformation of order $r = 45.26$ applied to the G-L quadrature integrating $p_2(x)$ in (0,1) at double precision with $n = 64$ samples.

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 Sub-Section 1.2.2, the former requires $n = \frac{\lambda_{max}+1}{2} = 18$ nodes to achieve double precise integration however, we immediately realise that our library (more sepcifically the monomial transformation quadrature rule) outperforms the classical G-L quadrature rule for a fixed double precision accuracy as only $n = 12$ samples are needed. We also remark that the transformation order needed by the monomial map is significantly lower than the previous two cases, in particular less than 1.

Since $p_3(x)$ does not have a singularity at $x = 0$, the processing of G-L quadrature yields a monomial transformation quadrature with samples that are not clustered on $x = 0$. In this instance instead, the transformation forces the performance of the quadrature to effectively integrate 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.

**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))    = 0.0833333333333333 **
 ** I_n(p(x)) = 0.0833333333333326    [with parameters in float128 precision] **
 ** E_n(p(x)) = 8.8727918916220005e-17  [with parameters in float128 precision] **
 ** I_n(p(x)) = 0.0833333333333323    [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>
```

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.

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

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

Chapter 3

# USER INTERFACE

Now that we have introduced the fundamentals regarding execution and interaction with the library, in this chapter we hereby expose its I/O operations interface by quickly building a new custom application. In the first section we discuss the input source file, amendable by the user to create her/his first application with the QUASIMONT. In the second section we address the outputs of the library and how the results are exported; we also consider the main differences between the loud and silent mode of execution. Subsequently we briefly outline how QUASIMONT's results can be used by other applications as they are imported in external code by integrating our library in a larger software. Finally we provide an exposition to the methods instantiated directly by the primary module 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 application by specifying the generalised polynomial $p_1(x)$ introduced in the test driver of Section 2.4. We can simply create a copy of the source file Main.cpp and of the associated top-level makefile located in **QUASIMONT/test** into a new subdirectory called **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. For the latter we must:

- substitute Test to MyApp whenever it is mentioned, renaming the application built by the compiler;
- alternatively define a macro variable EXE_NAME = MyApp and invoke it as $(EXE_NAME) .

For the former we need to specify the inputs related to the two sequences of coefficients and exponents that uniquely define the polynomial itself 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 throws an error message and exit the program. We shall retain the exact same sequence that has been previously defined in $p_1(x)$ as reported below

Main.cpp (input source file)

```cpp
#include "Quasimont.h"

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

  return 0;
}
```

We remark that the 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 following we first discuss how outputs are generated for the loud mode of execution of QUASIMONT for the MyApp application initialised in the previous section; we wrap the section up by exposing the different input parameters and outputs of QUASIMONT when executed in silent mode. 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 and lastly the value of the primitive (or analytic integral), numerical integral and the relative error obtained using the new quadrature parameters in quadruple (i.e. float128 f.p. format) and double precision. The classical G-L quadrature parameters are stored 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 between, float128 and double, with which to export the new quadrature nodes and weights to reach the prescribed relative precision. The optimality here is meant as the lowest-precision f.p. format that still allows to retain a machine-epsilon accuracy (we specified double f.p. precision however the procedure is easily generalised using C++ template parameters) for the relative error of the integral computed through the monomial transformation quadrature rule. The results of the quadrature and the new nodes and weights are then exported in three separate files called Results.txt, Nodes.txt and Weights.txt, all located in the **QUASIMONT/MyApp/output** subdirectory created automatically by the application. These text files collect the inputs and outputs of the library alongside the values of the numerical integral approximated using both the classical G-L and the proposed monomial transformation quadrature rule together with the exact analytical result. The file Results.txt is therefore intended for the user to have an immediate feedback on the quality of the approximation made by 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 2.2. The resulting new quadrature samples, established by the monomial transformation quadrature rule and optimised accordingly by the aforementioned routine, are exported with the f.p. precision that guarantees the a-posteriori relative error of the numerical integral to be within the machine-epsilon in double precision. When executing QUASIMONT in loud mode, as we did for the test drivers in Section 2.4 and for the present application, those new quadrature parameters are not streamed to the stack/heap as outputs of the primary method `quasimont`; they can only be retrieved manually (at any time) by the user by accessing the aforementioned text files. Furthermore, as explained extensively thus far, the user is required to fully specify the input generalised polynomial by providing a list of coefficients and exponents of each monomial term that constitute the integrand. However, as anticipated in the closing lines of Section 2.1, the primary module gateway's `quasimont` is overloaded to a silent mode that takes only two `double` parameters specifying $\lambda_{\min}$ and $\lambda_{\max}$ instead of the full sequence of exponents and coefficients. The silent mode then defaults both generalised monomials to have unitary coefficients and thus computes the quadrature of the following integrand

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

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

Main.cpp (silent mode execution)

```
#include "Quasimont.h"

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

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

  return 0;
}
```

The above execution of QUASIMONT in silent mode suppresses any on-screen printing of the results and generates a single `outputs.txt` file containing all the information regarding the monomial transformation quadrature rule (namely the values of $\lambda_{\min}$ and $\lambda_{\max}$ and the a-posteriori relative error) followed by the new same nodes and weights returned to memory. We emphasize however that, though the new nodes and weights written to `outputs.txt` are optimised by QUASIMONT's routine to be the proper f.p. format, those that are instead streamed to memory by the silent mode are, by default, returned in `double` precision regardless of the associated accuracy of the numerical integral. The reason for this choice of implementation for the silent mode of QUASIMONT is two-fold:

- on the practical side, the silent mode of QUASIMONT is providing the user with the core fundamentals of its functionality while stripping down the ancillaries concerning the testing, monitoring and debugging features s.a. on-screen information printing and progress tracking. Furthermore we realised that in most applications of interest in computational engineering it is frequent to only care about the extreme bounds of the input generalised polynomials, thus disregarding both all the real-valued exponents in between $\lambda_{\min} > -1$ and $\lambda_{\max}$ and also the values of the coefficients of each monomial term. As such, should the user be concerned with the convergence properties of the application we encourage her/him to go ahead instantiating the loud mode of `quasimont` as it always assures the optimal f.p. format for the quadrature nodes and weights. If convergence study is not a concern, then the silent mode should be used for its enhanced efficiency in I/O handling.
- rather than a placeholder type specifier (i.e. `auto`) to be deduced at compile time for exporting in memory the new nodes and weights in an optimised f.p. format we simply resort to a *matrix*-like structure in `double` precision. This choice was made to facilitate the interface of the user with our library as the `double` is much more wide-spread in use than the `float128` quadruple precision f.p. counterpart

## 3.3 STATIC-LIBRARY AND SOFTWARE INTEGRATION

While we demonstrated the effectiveness and consistency of QUASIMONT as a 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, in both its loud and silent modes, 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 located in the **QUASIMONT/build** 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
```

To showcase how to easily integrate the static library of our software in an external code we implemented a simple applications that is shipped with the library itself in **QUASIMONT/static**; by default the aformentioned make static rule will compile and 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 is -I../include;
- the actual libquasimont.a itself by adding path/to/QUASIMONT/static/libquasimont.a to the GCC linker option. For our case this path variable is that of the current directory.

Therefore, if the user wants to build an external App using the static version of 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 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

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

We recall that it is fundamental to correctly link all the third-party dependencies alongside QUASIMONT as reported in Section 2.1; failing to do so will result in undefined references. Moreover, the Quasimont.h header file needs to be included in the source code in order to avoid undefined references to QUASIMONT's methods. This is clearly exemplified in the snippet below in which we implement a simple procedure for taking quasimont primary module's inputs from terminal rather than an input source file discussed in Section 3.1. The user should input the coefficients and the exponents of the following generalised polynomial $p(x) = 0\,x^{-\frac{1}{2}} + x^{14} + 0\,x^{28}$; this is to showcase the ability of QUASIMONT of achieving much higher precision for the numerical approximation when the number of quadrature samples $n$ is overvalued with respect to the degree of the integrand.

App.cpp (user's external source code)

```cpp
#include "Quasimont.h"

int main()
{
  // TEST ON STATIC LIBRARY THIRD-PARTY INTEGRATION
  int n, counter = 0;
  float128 input;
  std::vector<float128> coeff_sequence, muntz_sequence;
  std::cout << "Insert the number of terms of the input polynomial: ";
  std::cin >> n;
  while(counter < n)
  {
    std::cout << "\n\nCoefficient: ";
    std::cin >> input;
    coeff_sequence.push_back(input);
    std::cout << "\nExponent: ";
    std::cin >> input;
    muntz_sequence.push_back(input);
    counter++;
  }
  quasimont(muntz_sequence, coeff_sequence);
  // DEFINE PARAMETERS FOR LOADING PROCEDURE
  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 FROM 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] **"
            << "\n\n ** E_n(p(x)) = " << fabs(In-1.0/(15.0))/fabs(1.0/(15.0)) << "  [with
    ↪ reloaded parameters in double precision] **" << std::endl;
  return 0;
}
```

The execution of this App.cpp with QUASIMONT's loud mode does not allow to keep our optimised nodes and weights in memory as they are instead exported via text files in a separate directory (see Section 3.2). In the App.cpp snippet above we implemented a simple and straightforward, automatised procedure that allows the user to load back in memory those parameters, from the text files, as std::vector< > data structures. In our case we re-imported them with float128 f.p. precision however this decision is arbitrary. Nevertheless we can use the re-imported nodes and weights to fulfill any of user's need, e.g. by recomputing the numerical integral in the last lines of the snippet above we showcase that no loss of information occurs when loading back the optimised quadrature parameters. As a matter of fact, as reported above, the integral of $p(x)$ computed and printed on-screen by QUASIMONT's loud mode and the one computed manually by re-loading the transformed quadrature parameters coincide exactly apart for a trailing numerical error due to representation of constant literals in float128 f.p. format.

Execution of external source code using QUASIMONT' static library

```
user@machine: home/QUASIMONT> cd static/
user@machine: home/QUASIMONT/static> make
g++  -c App.cpp -o App.o -ansi -std=c++17 -I../include
g++ App.o -o App libquasimont.a -lm -lquadmath -lgsl -lgslcblas
./App
Insert the number of terms of the input polynomial: 3

Coefficient: 0
Exponent: -0.5

Coefficient: 1
Exponent:   14

Coefficient: 0
Exponent:   28

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

 Input polynomial p(x) =  0*x^(-0.5) +x^(14)  0*x^(28)

 ** Accepted sequence of exponents **
    {-0.5, 14, 28}
 ** Lambda_min = -0.5, Lambda_max = 28 **
 --------------------------------------------------
 ** N_min = 52
 ** Beta_min = 3.51043403, Beta_max = 335.894726 **
 ** Transformation order = 10.3189638 **
 --------------------------------------------------
 ** Using double f.p. format for nodes and weights **
 ** I(p(x))    = 0.066666666666666667 **
 ** I_n(p(x)) = 0.066666666666666667    [with parameters in float128 precision] **
 ** E_n(p(x)) = 1.0291688140319292e-32  [with parameters in float128 precision] **
 ** I_n(p(x)) = 0.066666666666666664    [with parameters in double precision] **
 ** E_n(p(x)) = 4.4965745828494425e-17  [with parameters in double precision] **

 ** QUASIMONT HAS TERMINATED **

 ** I_n(p(x)) = 0.066666666666666665    [with reloaded parameters in double precision] **
 ** E_n(p(x)) = 7.8398326893941964e-18  [with reloaded parameters in double precision] **
```

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. 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. If such subdirectory is not placed properly the library fails 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 above App.cpp as it is located in sudirectory **QUASIMONT/static** that is on the same level of the aforementioned **QUASIMONT/data** .

## 3.4 MODULAR CALLS FOR INDEPENDENT INTEGRATION/DEVELOPMENT

In this section we explore in more details the methods that interact directly within the primary module quasimont. The purpose of this brief Section is to expose the fundamental modular calls that are required at run-time for the execution of the monomial transformation quadrature rule implemented in the template method quasimont. By explicitly outline the scope and I/O data structures handled at run-time we intend to provide the user with a capability of using some, or all of those methods independently from the execution or integration of our library. The user can find the types that have to be defined to correctly instantiate each of the following methods. We refer the user to Chapter 4 for the description of those methods where we report the comment block proceeding each function definition. With reference to the snippet reported in Section 2.1, the Quasimont.cpp source file instantiates 5 of the 12 total methods that constitute the library, which are specified in the following list sorted by the order of instantiation:

- **input_data** = manageData (**muntz_sequence**, **coeff_sequence**)
  Interfaces with user's inputs provided in the main of the application by arranging the proper data-structures (see Sub-Section 4.2.1).
  - **INPUTS** :
    - std::vector<T> **muntz_sequence** is a list of real numbers $\{\lambda_0, \lambda_1, \ldots, \lambda_k\}$ representing the terms of a truncated Müntz sequence of real-valued exponents associated to the generalised polynomial $p(x) \in \mathbb{P}_{\lambda_{\max}}$, $\lambda_{\max} = \max_k\{\lambda_0, \lambda_1, \ldots, \lambda_k\}$ that the user wants to integrate using the monomial transformation quadrature rule. The data type T templatising the structure is one amongst the discussed f.p. formats float128 or double;
    - std::vector<T> **coeff_sequence** is a list of real numbers $\{c_{\lambda_0}, c_{\lambda_1}, \ldots, c_{\lambda_k}\}$ representing the coefficients of the terms of the generalised polynomial. The order of the elements and data type T of the structure has to match that of **muntz_sequence**.
  - **OUTPUTS** :
    - std::tuple<int, std::vector<float128>> **input_data** contains an int variable $n_{\min}$ and a $2-$ dimensional std::vector<float128> array $\{\lambda_{\min},\ \lambda_{\max}\}$; the former is the minimum number of quadrature nodes computed as the sole real root of (1.18) and the latter is the infimum and supremum of the **muntz_sequence** input list.

- **monomial_data** = streamMonMapData (**n_min**)
  Computes and extracts all the necessary parameters to uniquely specify and build the monomial map, i.e. its order $r$ (see Sub-Section 4.2.2).
  - **INPUTS** :
    - int **n_min** is one of the outputs of manageData, specifically $n_{\min}$, which is easily accessed with std::get<0> (**input_data**).
  - **OUTPUTS** :
    - std::tuple<int, std::vector<float128>> **monomial_data** contains an int variable $n_{\min}$ and a $2-$ dimensional std::vector<float128> array $\{\beta_{\min},\ \beta_{\max}\}$; the former is the closest greater even integer of the minimum number of quadrature nodes provided with **n_min** while the latter is a $2-$dimensional array containing the entries of TabulatedErrorValues.csv for the minimum and maximum monomial exponents integrated exactly with $n_{\min}$ quadrature nodes.

- **transf_order** = computeMapOrder (**lambdas**, **betas**)
  Using the parameters streamed by the above method, compute the transformation order $r$ of the monomial map for the input generalised polynomial (see Sub-Section 4.1.3).
  - **INPUTS** :
    - std::vector<float128> **lambdas** is one of the outputs of manageData, specifically $\{\lambda_{\min},\ \lambda_{\max}\}$ and they are accessed with std::get<0> (**input_data**);

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

■ **OUTPUTS** :

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

▶ **quad_data** = computeQuadParams (**transf_order**, **n_min_even**)
Based on the monomial map computed at the previous step (specified by its order $r$) derives the computational parameters, i.e. nodes and weights, of the monomial transformation quadrature rule (see Sub-Section 4.1.4).

■ **INPUTS** :

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

■ **OUTPUTS** :

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

▶ optimiseData (**quad_data**, **muntz_sequence**, **coeff_sequence**)
This method optimises the f.p. format ( float128 or double ) of the monomial transformation quadrature rules' parameters by computing the numerical integral in (1.14) and the associated a-posteriori relative error in (1.7) (see Sub-Section 4.2.3), that are displayed at run-time.

■ **INPUTS** :

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

■ **OUTPUTS** :

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

## 3.5 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 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 update
user@machine: home> sudo apt-get install libgmp3-dev libmpfr-dev gnuplot
user@machine: home> # For RPM-based distros (CentOS, Fedora, SUSE, Scientific Linux ...)
user@machine: home> sudo dnf makecache --refresh
user@machine: home> sudo dnf install gmp-devel mpfr-devel gnuplot
```

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 are 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.

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;`
  - `computeQuadParams;`
  - `computeQuadRule;`
  - `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: there might be cases in which the user wants to integrate generalised
//                 monomials rather than the alike polynomials. In those cases the
//                 monomial transformation quadrature rule requires additional input to
//                 work properly. The library handles such case by either allowing the
//                 user to manually input the exponent of the additional term (from the
//                 CLI) or by letting the user specify the number of nodes to be used.
//                 In this last instance this method automatically computes the resulting
//                 maximum exponent of an additional term (lambda_max) that the monomial
//                 transformation quadrature rule can precisely integrate.
//
/////////////////////////////////////////////////////////////////////////////////////////

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: this method solves equation (1.18) in doc/UserManual.pdf, which is a
//                 7-th degree polynomial in n_min (the number of quadrature nodes).
//                 Of the 7 roots, it then extracts the only real one and then takes its
//                 ceil to return an integer value for n_min. The solver itself is a
//                 class' method implemented in the 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: this method computes the order of the monomial transformation (1.16)
//                 in doc/UserManual.pdf. It is derived as a linear interpolation of the
//                 minimum and maximum bound of inequality (1.17) of doc/UserManual.pdf.
//
////////////////////////////////////////////////////////////////////////////////////////

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

### 4.1.4 computeQuadParams

MonMap.cpp/computeQuadParams

```
////////////////////////////////////////////////////////////////////////////////////////
//
//       FUNCTION: [{new_x, new_w, old_x, old_w}] = computeQuadParams(r, n_min)
//
//          INPUT: - r = output of function 'computeMapOrder'
//                 - n_min = output of function 'streamMonMapData'
//
//         OUTPUT: - {new_x, new_w} = new set of monomial transformation quadrature rule
//                                    nodes (x) and weights (w)
//                 - {old_x, old_w} = old set of classical G-L quadrature nodes (x) and
//                                    weights (w) following the affine map
//
//    DESCRIPTION: with the transformation order being set, the monomial map is applied
//                 to transform the G-L nodes and weights, previously mapped from (-1,1)
//                 to (0,1) via the affine (linear) map in (1.12) in doc/UserManual.pdf.
//                 The complete set of new and old nodes and weights are collected in a
//                 tuple and returned.
//
////////////////////////////////////////////////////////////////////////////////////////

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 computeQuadRule

MonMap.cpp/computeQuadRule

```
//////////////////////////////////////////////////////////////////////////////////////
//
//       FUNCTION: In = computeQuadRule(x, w, muntz_sequence, coeff_sequence)
//
//          INPUT: - x = output of 'computeQuadParams' as optimised by 'optimiseData'
//                 - w = output of 'computeQuadParams' as 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: the computation of the numerical integral using a quadrature formula
//                 involves the sum of the weighted values that the integrand assumes on
//                 specified samples of the interval. This method uses such information
//                 to implement the weighted sum and returns it. Note that this method
//                 does not implement a specific quadrature rule, but rather a generic
//                 one. In the context of the library it is therefore used for both the
//                 monomial transformation and the classical G-L quadrature rules.
//
//////////////////////////////////////////////////////////////////////////////////////

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

### 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
//                                                               'computeQuadRule'
//                 - 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 above method
//                 computeQuadRule for a specific quadrature rule and let I be the exact
//                 (analytic) counterpart of such integral. The relative error of the
//                 quadrature can be computed a-posteriori as R_n = |I - I_n|/|I| as in.
//                 (1.7) of doc/UserManual.pdf. This method is the implementation of such
//                 error calculation 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, by.
//                 two unique lists of coefficients and exponents. This method first
//                 checks the proper form of those inputs. In particular:
//                    - the number of exponents and the number of coefficients coincide;
//                    - the input polynomial is at least a binomial (otherwise the
//                      further user-input as reported in doc/UserManual.pdf);
//                    - lambda_min > -1 (otherwise the input sequence of exponents is
//                      not a Muntz sequence and the polynomial is not L1);
//                 Following the checks the exponents' sequence is then sorted locally
//                 to identify and return lambda_min/lambda_max at global level alongside
//                 the associated minimum number of nodes computed n_min.
//
/////////////////////////////////////////////////////////////////////////////////

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 'computeQuadParams'
//                 - muntz_sequence = sequence of real exponents of the polynomial
//                 - coeff_sequence = sequence of real coefficients of the polynomial
//
//         OUTPUT: no outputs
//
//    DESCRIPTION: the method automatically selects the most optimised format possible
//                 (between double and float128) to export the output results, i.e. the
//                 transformed (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 f.p. format with lowest
//                 precision that still allows to retain a machine-epsilon accuracy (we
//                 specified double-precision however the procedure is easily generalised
//                 using the method's templatisation) for the relative error of the
//                 integral computed using the monomial transformation quadrature rule.
//
/////////////////////////////////////////////////////////////////////////////////////////

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 'computeQuadParams' optimised by
//                                 'optimiseData'
//                 - optim_weights = output of function 'computeQuadParams' optimised by
//                                 'optimiseData'
//                 - [I_new, E_new, I_old, E_old] = values of the numerical quadratures
//                                  (I_new, I_old) and relative errors (E_new, E_old)
//                                  obtained using the nodes and weights provided
//                                  by the monomial transformation and classical G-L
//                                  quadrature rules respectively.
//
//         OUTPUT: no outputs
//
//    DESCRIPTION: once the monomial transformation quadrature rule is completed, data is
//                 streamed in output and written to the appropriate text files. The 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 limiting numerical
//                 cancellation errors.
//
//////////////////////////////////////////////////////////////////////////////////////////

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++ Monomial Transformation Quadrature Rule for High Precision Integration of Singular and Generalised Polynomials”. 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`.