Advanced Programming for Scientific Computing Project

Tree Approximation for hp-Adaptivity

Implementation of binary tree approximation algorithm for high order mesh
hp-adaptation

Simone Carriero
ID 864234

Academic year 2017–2018
Abstract

This report describes an implementation of the \( hp \)-refinement algorithm proposed in \cite{1} to obtain a near optimal approximation performances for the solution of the problem of approximation of a function on a given domain.

A mathematical analysis of the algorithm and of the expected performances is carried on, and validity of the obtained theoretical results is verified on a set of sample problems.

The report describes also the main design choices made for the development, and provides a user manual allowing both usage of the software on different test cases, and its extension to include different approximation functions, mesh managing libraries, and higher dimensions cases.
Contents

Introduction 1

1 The Mathematics of the Problem 3
  1.1 Algorithm introduction 3
  1.2 Adaptation algorithm 7
  1.3 Element bisection 8
  1.4 Error evaluation 9
    1.4.1 Numerical integration 9
    1.4.2 The mapping process 10
    1.4.3 Function projection 11
  1.5 Theoretical results 12
    1.5.1 Analytic functions 13
    1.5.2 Singular functions 13
    1.5.3 2D case 14

2 The Software 15
  2.1 Requirements 15
  2.2 Architectural Design 18
    2.2.1 The main algorithm implementation 18
    2.2.2 The Abstract Factory pattern implementation 19
    2.2.3 The Plugin pattern implementation 20
    2.2.4 The meta Factory implementation 22
    2.2.5 Working with the mesh 24
    2.2.6 The Strategy pattern usage 25
    2.2.7 The Flyweight pattern usage 26
    2.2.8 Collecting statistics on algorithm execution 26
    2.2.9 Managing configuration files 28
  2.3 Coding details 29
    2.3.1 Quadrature Rules 29
    2.3.2 Mapping process 30
    2.3.3 Finite Elements 31
    2.3.4 Binary Elements 34
    2.3.5 The Algorithm 35
Contents

2.4 Design Justification ................................................. 37
  2.4.1 Mesh library selection ....................................... 37
  2.4.2 Template vs polymorphism .................................... 38

3 Numerical Tests ...................................................... 41
  3.1 1D mesh .......................................................... 41
    3.1.1 Polynomial \( f(x) \) ........................................ 41
    3.1.2 Piecewise polynomial \( f(x) \) ............................. 42
    3.1.3 Analytic \( f(x) \) ........................................... 43
    3.1.4 Piecewise analytic \( f(x) \) ............................... 44
  3.2 2Dmesh ............................................................ 47

4 User manual ........................................................... 49
  4.1 Software description .............................................. 49
  4.2 Setup and initialization ......................................... 49
  4.3 Nominal operations .............................................. 51
  4.4 Error conditions ................................................. 52
  4.5 Non nominal operations .......................................... 55

Conclusions ............................................................ 59

Bibliography ............................................................ 61
List of Algorithms

1.1 hp refinement process .................................................. 6
1.2 hp-Refinement Algorithm ............................................. 7
1.3 Trim algorithm ........................................................... 8
Listings

2.1 Parsing variadic arguments .................................................. 27
2.2 Interface for a quadrature rule .............................................. 30
2.3 Interface for a map ............................................................. 30
2.4 Attributes of an affine map .................................................. 31
2.5 Interface for a finite element ............................................... 32
2.6 Interface for the std finite element ....................................... 32
2.7 Attributes of a binary element .............................................. 34
2.8 Interface for mesh refiner .................................................... 35
2.9 Attributes of the mesh refiner .............................................. 36
2.10 The ghost node implementation ........................................... 36
2.11 The refinement algorithm implementation .............................. 36
4.1 Register different mesh refiner ............................................. 56
4.2 Register different quadrature rule ........................................ 57
**Introduction**

In the solution of PDE problems, mesh adaptation plays an important role as a key to speed up numerical simulations without losing solution accuracy. Facing the even more general problem of the approximation of a generic function on a domain $\Omega$, there are two main approaches to adaptivity that can be followed. The first one, called $h$-refinement, consists in reducing the size $h$ of selected elements $\Delta$ of the current partition of $\Omega$; the second one, called $p$-refinement, considers instead the same partition, and improves the approximation exactness increasing the approximation order $p(\Delta)$ on selected elements $\Delta$.

This logically raises the question on what’s the best way to combine these two operations, that is $hp$-adaptivity, in such a way that the obtained partition gives the best possible approximation performances. In this regard, a near-optimal algorithm has been proposed by [1] which, fixed the number of refinements that can be operated on the partition, suggests a strategy to distribute them among the two possible aforementioned approaches.

Taking this result as starting point of our work, the aim of this project is to go beyond the theoretical approach given by [1], and so to build a software suite that allows us first to numerically verify the performances of the developed algorithm, and to set its range of applicability to practical cases; then the software built in this way will provide a possibly external user with any necessary tool to apply with minimum effort the refinement algorithm, and so to exploit its potential, to a wide range of problems, i.e. to the Discontinuous Galerkin solution of PDE equations. With this perspective we have developed an open source C++ toolkit that provides an implementation of the algorithm in literature, written keeping in our mind generality and versatility as basic principles. The C++ suite is released together with a detailed manual, to help the user in program learning, usage and in possible third parties developments, too.

The remaining of this report is organized as follows:

- In Chapter 1 we will depict the problem from a mathematical point of view. We will give the explicit theoretical picture of the algorithm at an abstract level, studying its range of applicability and the hypothesis it is build on. Moreover the mathematical details of all the elements it is composed of, needed to apply it to practical cases, will be illustrated. Then we will make a theoretical analysis of the expected results and performances, that will be used in successive
sections to test the algorithm implementation.

- In Chapter 2, we will describe the C++ implementation of the algorithm in the delivered software, focusing on the programming choices we have made during software development. We will explain and justify the programming philosophy that has led the coding process, illustrating both the general design the software has been built on, and the specific implementation in main classes of the mathematical tools previously reported at abstract level.

- In Chapter 3, we will report the results of the numerical tests we have run on problems that have been studied in first chapter from a theoretical point of view. The purpose of this section will be to show what we have done in order to check the correctness of the software programming in each of its components and, exploiting previously shown a priori results made available by the theoretical analysis, to validate the given implementation of the algorithm in the software, and so the algorithm itself.

- Chapter 4 constitutes the usage manual of the released library for the external user. It includes all the instructions to properly build and install the package on the system, together with a detailed description, and related solution instructions, of as many as possible runtime errors the user may face-off executing the program. Moreover we furnish tutorial on how to extend the code, to include support (expected but not yet coded) to external libraries different from the ones considered until now, or to exploit software components for use different from the main one intended in this release.
1 The Mathematics of the Problem

In this section we will briefly present the generic algorithm for the $hp$-refinement process of a mesh, described in detail in [1], and the mathematical structure adopted to give it a practical implementation. The optimality results given by [1] will be reminded, together with the test cases for which the optimality condition has been already studied from a theoretical point of view, therefore justifying the approach adopted to validate the algorithm itself and its implementation through numerical simulations, deeply described in next sections.

Only the final picture of the algorithm will be given, together with an explanation of the main quantities introduced; for a complete treatment, or for additional justification of the choices made, the reader may refer to [1].

1.1 Algorithm introduction

In the context of an $hp$-adaptive approximation of a function on a numerical domain, given a fixed value $N$, the problem reads as finding a near-best binary tree $T$ of complexity $N$, where the complexity of a tree is given by the sum of polynomial orders $p(\Delta)$ of its leaves $\Delta$, which represent the elements of the partition. Under the binarity assumption for $T$, an iterative procedure can be built that attaches to each leaf $\Delta$ a ghost tree with $p(\Delta)$ leaves, in such a way that the generated full binary tree $\mathcal{T}$ can be used as a proxy of $T$ for assembling $hp$-adaptive procedure (see figure 1.1).

![Figure 1.1: An example of tree $T$ and ghost tree $\mathcal{T}$](image)

Figure 1.1: An example of tree $T$ and ghost tree $\mathcal{T}$
1. The Mathematics of the Problem

Note that the algorithm is meant to be applied to a generic partition, so it is not tied to the geometry of the problem; however the binarity assumption requires that the $h$-refinement of a mesh element has to generate no more than two children elements at each application.

During the treatment $\mathcal{L}(T)$ will denote the set of leaves of $T$, so that the complexity will be $N = \#\mathcal{L}(T)$. Calling $e_p(\Delta)$ the local error on the element $\Delta$, where $p = \mathcal{P}(\Delta)$ is the order of the polynomial space used in the approximation at $\Delta$, the global error on the partition will be

$$\mathcal{E}(T, \mathcal{P}(T)) := \sum_{\Delta \in \mathcal{L}(T)} e_p(\Delta)$$

(1.1)

where

$$\mathcal{P}(T) := (\mathcal{P}(\Delta))_{\Delta \in \mathcal{L}(T)}$$

(1.2)

are the $p(\Delta)$ assignments.

So, given $N$ and suitable $e_p(\Delta)$ (see [1]), the aim is to build an iterative procedure which produces a pair $(T_N, \mathcal{P}(T_N))$ of a tree and corresponding polynomial orders satisfying a near-best $hp$-adaptive approximation condition, namely

$$\mathcal{E}(T_N, \mathcal{P}(T_N)) < C_1 \sigma_{c_2 N}$$

(1.3)

where

$$\sigma_N := \inf_T \inf_{\#\mathcal{P}(T) \leq N} \mathcal{E}(T, \mathcal{P}(T))$$

(1.4)

is the best approximation error for $N$ complexity trees.

In particular these results applies to the $L^2$-norm case, so that $e_p(\Delta)$ will be the $L^2$ integral of the difference between the function and the approximating polynomial. The procedure splits into two steps, the first one for just the $h$-refinement case, then its extension to the $hp$ one.

$h$-Refinement strategy

In the $h$-refinement case all elements will have the same polynomial order, so that the algorithm reduces to the identification of the element to be split at each iteration. In order to satisfy 1.3 taking the error $e(\Delta)$ as leading parameter for the refinement does not work (see [1]). Because of this, it is introduced a new quantity $\tilde{e}(\Delta)$ defined

$\tilde{e}(\Delta) = 0$ if both $e(\Delta)$ and $\tilde{e}(\Delta^*)$ are zero.
1.1. Algorithm introduction

as

\[ \tilde{e}(\Delta) := \frac{e(\Delta)\bar{e}(\Delta^*)}{e(\Delta) + \bar{e}(\Delta^*)} \quad (1.5) \]

\[ \tilde{e}(R) := e(R) \]

where \( \Delta^* \) is the father element of \( \Delta \) and \( R \) is the root of the tree.

So, starting from \( T_1 := R \), and refining the leaf with the highest \( \tilde{e}(\Delta) \) to pass from \( T_N \) to \( T_{N+1} \), it can be proven \((\text{[1]})\) that at each iteration the \( T_N \) tree will satisfy the condition:

\[ \mathcal{E}_h(T_N) \leq \frac{N}{N-n+1} \sigma_h^N \forall n \leq N \quad (1.6) \]

which is of the kind of \( \text{[1.3]} \), where the global error \( \mathcal{E}_h(T_N) \) and the best \( N \)-tern \( h \)-adaptive approximation error \( \sigma_h^N \) are defined as

\[ \mathcal{E}_h(T) := \sum_{\Delta \in \mathcal{L}(T)} e(\Delta) \quad (1.7) \]

\[ \sigma_h^N := \min_{\# \mathcal{L}(T) \leq N} \mathcal{E}_h(T) \quad (1.8) \]

**hp-Refinement strategy**

Starting from the previous step, the algorithm can be extended to allow higher polynomial orders on the elements of the binary tree. So, let’s call \( \Delta' \) and \( \Delta'' \) the children of the element \( \Delta \), and introduce the local \( hp \)-error \( E(\Delta) = E(\Delta, T_N) \)

\[ E(\Delta) := e_1(\Delta) \quad \text{if } \Delta \in \mathcal{L}(T_N) \]

\[ E(\Delta) := \min \{ E(\Delta'), E(\Delta''), e_p(\Delta) \} \quad \text{otherwise} \quad (1.9) \]

and, similarly to the \( h \)-refinement case, the quantity \( \tilde{E}_j(\Delta) \)

\[ \tilde{E}_j(\Delta) := \frac{E_j(\Delta)\tilde{E}_{j-1}(\Delta)}{E_j(\Delta) + \tilde{E}_{j-1}(\Delta)} \quad \text{if } j > 1 \]

\[ \tilde{E}_1(\Delta) := \tilde{e}(\Delta) \]

The \( E(\Delta) \) quantity will be needed to extract the tree \( T_N \) from the full binary tree \( T_N \), while the quantity \( \tilde{E}_j(\Delta) \) will lead the iterative construction of \( T_N \).

Once that \( T_N \) is given, getting \( T_N \) from it exploits a recursive algorithm which,

\[ ^2 \tilde{E}_j(\Delta) = 0 \text{ if both } E_j(\Delta) \text{ and } \tilde{E}_{j-1}(\Delta) \text{ are zero.} \]
starting from $T_N$, trims it at $\Delta$ every time $E(\Delta) = e_P(\Delta)(\Delta)$, so that the obtained $T_N$ is the minimal tree for which $E(\Delta) = e_P(\Delta)(\Delta)$ at all its leaves\footnote{Remember that the leaves of $T_N$ represent the mesh elements.}

Two last functions $q : T_N \rightarrow \mathbb{R}_+$ and $s : T_N \rightarrow \mathcal{L}(T_N)$ have to be introduced to define the $hp$-adaptive procedure: $s(\Delta)$ will give the information about the next to be $h$-refined leaf of the subtree rooted at $\Delta$, based on the value of the $q(\Delta)$ quantity. They are recursively defined as

\[
q(\Delta) := \tilde{E}_1(\Delta) = \tilde{e}(\Delta) \quad \text{if } \Delta \in \mathcal{L}(T_N) \\
q(\Delta) := \min\left\{ \max \left\{ q(\Delta') , q(\Delta'') \right\} , \tilde{E}_P(\Delta) \right\} \quad \text{otherwise}
\]

and

\[
s(\Delta) := \Delta \quad \text{if } \Delta \in \mathcal{L}(T_N) \\
s(\Delta) := s(\arg\max \left\{ q(\Delta') , q(\Delta'') \right\}) \quad \text{otherwise}
\]

and so, starting from $T_1 = \mathcal{R}$, at each iteration $N < N_{\text{max}}$ the tree $T_{N+1}$ is generated by subdividing the leaf $\Delta = s(\mathcal{R}) \in \mathcal{L}(T_N)$ and consequently incrementing by one its order $P(\Delta)$, then recursively increment by one the order for its father; in this way, every element of $T_N$ will have an order $P(\Delta)$ equal to the number of leaves of the subtree rooted at it, and so every trim on the full binary tree will produce a tree $T_N$ of complexity $N$.

\textbf{Algorithm 1.1} hp refinement process

1. Select unoptimal leaf
2. Bisect selected leaf
3. Compute interpolation error
4. Update tree
5. Make optimal trim

It can be proven (\cite{1}) that this $hp$-adaptation procedure provides near-best approximation and satisfies the following:

\[
\mathcal{E}(T_N, T_N) \leq \frac{2N - 1}{N - n + 1} \sigma_n 
\]

$\forall n \leq N$, which is again of the type 1.3
1.2 Adaptation algorithm

Remembering that $\Delta'$ and $\Delta''$ are the two children of the generic element $\Delta$, and $\Delta^*$ its father, the detailed description of the algorithm follows:

**Algorithm 1.2 hp-Refinement Algorithm**

1. $N \leftarrow 1$
2. $T_1 := \{R\}$,
   $\tilde{e}(R) := e(R)$,
   $E_1(R) := e(R)$,
   $\tilde{E}_1(R) := \tilde{e}(R)$,
   $q(R) := \tilde{e}(R)$,
   $s(R) := R$
3. while $N < N_{\text{max}}$ do
   4. $\Delta_N \leftarrow s(R) \triangleright$ selecting the less optimal leaf
   5. bisect $\Delta_N$
   6. for $\nabla \leftarrow \Delta_N', \nabla \leftarrow \Delta_N''$ do
      7. $\tilde{e}(\nabla) = \frac{e(\nabla)}{e(\nabla) + e(\Delta_N)}$
      8. $E_1(\nabla) = e(\nabla)$
      9. $\tilde{E}_1(\nabla) = \tilde{e}(\nabla)$
      10. $q(\nabla) = \tilde{e}(\nabla)$
      11. $s(\nabla) = \nabla$
   7. end for
   8. $\Delta \leftarrow \Delta_N$
   9. repeat \triangleright updating the tree
   10. $P(\Delta) = P(\Delta) + 1$
   11. compute $e_{P(\Delta)}(\Delta)$
   12. $E(\Delta) \leftarrow \min \{E(\Delta'), E(\Delta''), e_{P(\Delta)}(\Delta)\}$
   13. $\tilde{E}_{P(\Delta)}(\Delta) = \frac{E(\Delta)E_{P(\Delta)-1}(\Delta)}{E_{P(\Delta)}(\Delta) + E_{P(\Delta)-1}(\Delta)}$
   14. $D \leftarrow \arg \max \{q(\Delta'), q(\Delta'')\}$
   15. $q(\Delta) \leftarrow \min \{q(D), \tilde{E}_{P(\Delta)}(\Delta)\}$
   16. $s(\Delta) \leftarrow s(D)$
   17. if $\Delta \neq R$ then
      18. $\Delta \leftarrow \Delta^*$
   19. end if
   20. until $\Delta \neq R$
   21. end repeat
22. Trim($R$)

where the algorithm for the trim is simply given by 1.3

The algorithm can be easily extended to the case of a starting mesh with more than one element. Every element of the mesh can be considered the root $R_i$ of a tree of the kind of $T$, and the operation 4 of the algorithm can be replaced by $\Delta_N \leftarrow \arg \max_i \{q\ (s(R_i))\}$. Also stopping criterions different from the max iteration
1. The Mathematics of the Problem

Algorithm 1.3 Trim algorithm

```
procedure TRIM(Δ)  
    if E(Δ) == e_P(Δ) (Δ) then
        return 
    else
        TRIM(Δ′)
        TRIM(Δ″)
    end if
end procedure
```

one can be easily added. The typical error condition, which in this case reads

\[ E(T_N, P(T_N)) < \epsilon \]

can be added in the while statement, taking care that the trim operation in this case has to be done at each iteration of the while loop, since to compute the error the tree \( T_N \) is needed (see 2.11 for an example of implementation of this case).

1.3 Element bisection

The algorithm just described is based on a binary tree structure: this implies that the \( h \)-refinement of an element cannot generate more than two children elements. While this is obvious for the monodimensional case, where elements are intervals, it is not so common in the multidimensional case. For example, in the case of a 2D mesh with triangular elements there is more than one way to refine one leaf of the tree, as shown in 1.2. This in general may be due to the implementation of the refinement process given by the tool in use, or it can be done on purpose to get meshes with particular features (e.g., absence of hanging nodes), that are more difficult to manage and not supported by many mesh generators. Anyway, the requirement of a binary refinement limits the range of applicability of the algorithm in the multidimensional case, therefore to treat the bidimensional problem triangular elements have been chosen, with the bisection performed using the simple procedure, shown in 1.2b, that is:

- Select the refinement edge \( E(T) \), that is the longest edge of the triangle \( T \), and its opposite vertex \( v(T) \);
- Connect \( v(T) \) with the midpoint of \( E(T) \);
- The rised children \( T_1 \) and \( T_2 \) will have common vertex \( v(T_1) = v(T_2) \), the newly created vertex, and opposite refinement edges \( E(T_1) \) and \( E(T_2) \).

The 3D case have not been treated yet, its study is left for future development.
1.4 Error evaluation

In order to execute the algorithm, a formula for $e_p(\Delta)$ has to be given, ensuring that it fulfills the requirements necessary to guarantee the near-optimality condition on the error (see [1,4]). The choice was made to use the $L^2$-norm which, as already seen, is suitable for our purpose. So the formula for the approximation error of a function $u$ reads:

$$e_p(\Delta) = \int_{\Delta} (u - \pi_p)^2 d\Omega \quad (1.11)$$

where $\pi_p$ is the projection of $u$ over the space $P_p(\Delta)$ of polynomials of degree at most $p$ on $\Delta$.

When it comes to numerical evaluation of (1.11) we are confronted with two tasks: the computation of $\pi_p$ and the evaluation of the integral.

1.4.1 Numerical integration

For the numerical evaluation of integrals a quadrature rule is needed. Since for our purposes there is no need to have quadrature nodes on the boundary of the elements, a Gaussian quadrature has been chosen, in order to exploit its optimal performances (4). The choice is not binding, so that the selected quadrature can be replaced, without loss of generality, by a different rule. Gaussian nodes $\hat{x}_i$ are defined on the [-1,1] reference interval; the nodes $x_i$ on the generic interval can be obtained by mapping the reference nodes through an affine map (see 1.4.2).

For the 2D case different solutions can be applied, depending on the elements geometry: for squared elements optimal quadrature nodes can be recovered by tensorization of the 1D counterparts, with quadrature weights obtained as the product of the corresponding 1D weights; in the triangular case quadrature nodes on a standard triangle can be obtained by mapping the just mentioned nodes on the standard

4or in general for elements that can be mapped to the [-1,1] $\times$ [-1,1] square through an affine map
1. The Mathematics of the Problem

Figure 1.3: Transformation of the reference triangle $\hat{T}$ into the reference square $\hat{Q}$.

The slanting segments are transformed into vertical segments through the singular map

$$
(x_1, x_2) \rightarrow (\xi_1, \xi_2), \quad \xi_1 = 2 \frac{1 + x_1}{1 - x_2} - 1, \quad \xi_2 = x_2
$$

represented in 1.3, which is a bijection between $\hat{T}$ and $\hat{Q}$. Its inverse is given by

$$
(\xi_1, \xi_2) \rightarrow (x_1, x_2), \quad x_1 = \frac{1}{2} (1 + \xi_1) (1 - \xi_2) - 1, \quad x_2 = \xi_2
$$

This procedure leads to numerical instability towards the map singularity. For this reason in the $2D$ case, where the mesh will be composed of triangular elements, as seen in 1.3, the Dunavant rule [2] has been exploited. This choice guarantees a more stable evaluation of the integrals, even if, unlike the Gaussian rule, it is affected by the limitedness of the exactness order.

1.4.2 The mapping process

As briefly introduced in previous paragraph, for each considered geometry quadrature nodes are given on a standard element $\hat{\Delta}$, and basis functions too, as we will see in next section 1.4.3. Being $\Delta$ the generic mesh element, it can be defined the map $\phi : \hat{\Delta} \rightarrow \Delta$, whose explicit formula depends on the geometry of $\Delta$, to map each point of the standard element into one of the generic element. Since $\hat{\Delta}$ and $\Delta$ have the same geometry, the map $\phi$ will be linear and bijective, so that it can be represented as:

$$
\phi(x) = Ax + b
$$

with non singular $A \in \mathbb{R}^{n \times n}$ and $b \in \mathbb{R}^n$, being $n$ the dimensionality of $\Delta$.

E.g., in the monodimensional case, the map $\phi(x) : (-1,1) \rightarrow (a,b)$, with $\hat{\Delta} :=$

\[\text{Quadrature nodes and weights are tabulated, so the maximum exactness order reachable is binded by the tabulation.}\]
1.4. Error evaluation

Figure 1.4: Triangular affine map

\((-1,1), \Delta := (a,b),\) reads:

\[
\phi(x) := \frac{b-a}{2} x + \frac{b+a}{2}
\]

In the bi-dimensional triangular case, where \(\tilde{\Delta}\) and \(\Delta\) are shown in figure 1.4, the affine map will be:

\[
\phi(x) := \begin{bmatrix}
\frac{x_2-x_1}{2} & \frac{x_3-x_1}{2} \\
\frac{y_2-y_1}{2} & \frac{y_3-y_1}{2}
\end{bmatrix} x + \begin{bmatrix}
\frac{x_2+x_1}{2} \\
\frac{y_2+y_1}{2}
\end{bmatrix}
\]

1.4.3 Function projection

Given a basis \(\{\psi\}_i^p\) for the space \(\mathbb{P}_p(\Delta)\), the function \(u_p \in \mathbb{P}_p(\Delta)\) can be written as:

\[
\pi_p(x) = \sum_{i=1}^{p} u_i \psi_i(x)
\]

so the problem of computing the projection \(\pi_p\) of the function \(u\) becomes that of finding the coefficients \(u_i\). They can be computed, by definition of projection w.r.t. the scalar product \((\cdot,\cdot)\), solving a problem of the type

\[
(u, \psi_j) = (u_p, \psi_j), \quad \forall j = 1, \ldots, p
\]

that is

\[
\sum_{i=1}^{p} u_i (\psi_i, \psi_j) = (u, \psi_j), \quad \forall j = 1, \ldots, p \tag{1.14}
\]

Using the \(L^2\) scalar product as \((\cdot,\cdot)\), and choosing an orthogonal basis in order to diagonalize the mass matrix of the problem 1.14, the \(L^2\) projection coefficients \(u_i\) are simply given by

\[
u_i = \frac{(u, \psi_i)}{\|\psi_i\|}, \quad i = 1, \ldots, p \tag{1.15}
\]

\(^6\)A different basis can be used without loss of generality
For the one dimensional case the Legendre polynomials \{L_i\} have been exploited to build the basis; by tensorization it can be recovered an orthogonal basis for the 2D squared case. For triangular elements the composition of 2D Legendre polynomials with the singular map will not work, since the so obtained basis is no longer orthogonal; in this case it has been used the warped tensor product basis (4)

\[ \phi_k (\xi_1, \xi_2) := \psi_{k_1} (\xi_1) \psi_{k_1, k_2} (\xi_2) \]  

with \( \psi_{k_1} (\xi_1) = J_{k_1}^{(0,0)} (\xi_1) \), \( \psi_{k_1, k_2} (\xi_2) = (1 - \xi_2)^{k_1} J_{k_2}^{(2k_1+1,0)} (\xi_2) \) for each pair of integers \( k = (k_1, k_2) \), and \( (\xi_1, \xi_2) \) given by the singular map 1.12

By now basis functions have been intended to be defined on standard elements: an orthogonal basis for the generic element is obtained by composing the standard basis functions with an affine map of the kind 1.13 which does not make them lose the orthogonality property.

The numerical integration described in 1.4.1 can be exploited to compute the \( L^2 \) products \((\cdot, \cdot)\) and the \( L^2 \) norms present in 1.15

### 1.5 Theoretical results

As already shown, the algorithm 1.2 has the near-optimality property expressed, from a quantitative point of view, by the equation:

\[ E (T_N, T_N) \leq \frac{2N - 1}{N - n + 1} \sigma_n \]

The relation states that the \( hp \)-refinement process leads to a final error equal to the one given by an optimal algorithm, but with the need of a higher number of iterations. So now the aim is to build test cases that allow to check this sub-optimal behaviour, as well as the correctness of the implementation that will be described in the next sections. Unfortunately, given an input complexity value \( N \), there are no known results available about how the final partition should be done, i.e. there isn’t any function for which it can be stated a priori the pair \( (T_N, P (T_N)) \) expected as output of the algorithm. However there is literature about the polynomial approximation of particular sets of functions, for which equations for the a priori control of the approximation error have been developed; even if there isn’t enough information about the final mesh structure, its sub-optimality can be checked verifying that the approximation error follows the expected theoretical profile as its complexity grows. For this purpose two cases have been taken into account, exploiting the results given by [5]: the one of the interpolation of an analytic function, and the one of a piecewise analytic function.

\[ J_i^{(\alpha, \beta)} \] denotes the family of Jacobi polynomials with parameters \( \alpha \) and \( \beta \); in particular \( J_i^{(0,0)} = L_i \) are the Legendre polynomials
1.5. Theoretical results

1.5.1 Analytic functions

Let \( \hat{\Omega} \) be the \([-1, 1]\) interval, define \( V^k(\hat{\Omega}) \) as the space of all \( u_0 \in L^2(\hat{\Omega}) \) for which the norm

\[
\|u_0\|^2 = \sum_{i=0}^{k} \int (1 - \xi^2)^i |u_0^{(i)}(\xi)|^2 d\xi
\]

(1.17)
is finite. Let then \( s_0 \) be the partial Legendre series

\[
s_0 = \sum_{i=0}^{p} a_i L_i(\xi)
\]

Then it can be proven (\cite{5}) the following asymptotic convergence estimate:

\[
\inf_{s \in \mathcal{S}^p} \|u_0 - s\|_{L^2(\Omega)} \leq C(k) p^{-k} |u_0|_{V^k(\hat{\Omega})}
\]

(1.18)

where \( \mathcal{S}^p = \{ u \in C^0 : u|_{\Omega_j} \text{ is polynomial of degree } p_j, \Omega_j \in \mathcal{T}, j = 1, ..., M(\mathcal{T}) \} \), with \( M(\mathcal{T}) \) the number of elements of the mesh \( \mathcal{T} \).

Applying this result to an \( hp \)-refinement process framework the following relations can be obtained (see \cite{5} for the details), considering \( N = \dim(\mathcal{S}^p(\Omega, \mathcal{T})) \):

\[
\|u_0 - s_0\|_{L^2(\Omega)} \leq CN^{-\min(p,k)-1}
\]

(1.19)

for the \( h \)-refinement case (fixed \( p \)), and

\[
\|u_0 - s_0\|_{L^2(\Omega)} \leq CN^{-k-1}
\]

(1.20)

for the \( p \)-refinement case (fixed \( h \)). For a regular \( u_0 \), that is high \( k \), is then convenient to increase \( N \), and so the complexity of the partition, by increasing \( p \). It turns out that for a smooth enough \( u_0 \) arbitrary high algebraic rates of convergence are possible for sufficiently high polynomial degree \( p \). If \( u_0 \) is analytic, exponential convergence is expected, and the following convergence estimate can be proven for the \( hp \)-refinement process (\cite{5}):

\[
\|u_0 - s_0\|_{L^2(\hat{\Omega})} \leq C(r) p^{-1} r^{-2p}
\]

(1.21)

for some \( r > 1 \), with \( s_0 = \pi_p u_0 \in \mathcal{S}^p, p = 1, 2, 3, ... \). 

1.5.2 Singular functions

The second family of functions that will be tested is the one of piecewise analytic functions. In this case \cite{5} gives an a priori estimate on the error too.

A mesh \( \mathcal{T} \) is called geometrically graded towards \( x_0 = 0 \) with grading factor

\footnote{This is referred to as spectral convergence}
1. The Mathematics of the Problem

\( \sigma \in (0, 1) \) if

\[ T = \{(x_{j-1}, x_j)\}_{j=1}^M, \quad x_j = \sigma^{M-j} \]

that is the mesh is \( h \)-refined near \( x_0 \). For a mesh such that and \( u_0 \) analytic in \( x \neq x_0 \), with the degree vector given by

\[ p_1 = 0, \quad p_j = \max\{0, [\mu_j]\}, \quad j = 2, ..., M \quad (1.22) \]

for some \( \mu(\sigma) \), it holds true that

\[ \inf_{s \in S_P} \|u_0 - s\|_{L^2(\hat{\Omega})} \leq Ce^{-b\sqrt{N}} \quad (1.23) \]

where \( C > 0, b > 0 \) are independent on \( N \).

So what is expected in this case is a finer mesh near the singularity, with increasing polynomial degree moving away from it.

1.5.3 2D case

The bidimensional problem is harder to treat than the 1D one, since the study on the approximation error is strongly correlated to the type of domain discretization adopted. A full treatment of this case goes beyond the scope of this work, also because a deeper study of the algorithm and testing of its implementation performances has to be planned for the 2D case. However, for the particular case of a polygonal domain discretized with a triangulation with possible hanging nodes, which is the framework depicted in 1.3, an estimate similar to 1.23 can be recalled (5):

\[ \inf_{s \in S^r(\Omega, T)} \|u - s\|_{L^2(\Omega)} \leq Ce^{-bN^{\frac{1}{4}}} \quad (1.24) \]

with \( C > 0, b > 0 \) independent on \( N \) and \( u \) analytic but possibly singular in \( x_0 = 0 \).
2 The Software

In this section we will describe the software system designed and developed in order to implement the concepts provided in previous section. Starting from the high level requirements that have driven the implementation, we will follow with a description of the design adopted, with a special focus on the design patterns used throughout the process, then a justification of the most important and committing choices made where different alternatives were in place.

2.1 Requirements

This section contains the requirements that constitute the software specifications that have driven the coding and implementation. Each of them is reported giving the following information:

1. Requirement Title
2. Requirement Description
3. Requirement Type: identifies the type of the requirement, as per the table 2.1
4. Requirement Verification Method: defines what kind of check will be made to verify that the requirement has been properly implemented by the software, as per the following list:

<table>
<thead>
<tr>
<th>Type</th>
<th>Classification</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>FUN</td>
<td>Functional Requirements</td>
<td>These specify <em>what</em> the software has to do. They define the purpose of the software.</td>
</tr>
<tr>
<td>OPE</td>
<td>Operational Requirements</td>
<td>These specify how the software will run and how it will communicate with the users.</td>
</tr>
<tr>
<td>DES</td>
<td>Design Requirements</td>
<td>These specify specific constraints the design shall satisfy.</td>
</tr>
<tr>
<td>RES</td>
<td>Resource Requirements</td>
<td>These specify constraints on usage of specific resources, like operating system or specific libraries.</td>
</tr>
</tbody>
</table>
2. The Software

a) TEST: Implementation of this requirement will be demonstrated by a specific test.

b) INSPEction: Implementation of this requirement will be demonstrated by some form of visual inspection (e.g., design, software code, documentation, etc).

5. Requirement Verification Evidence: points to evidence showing that the requirement has been correctly implemented.

Follows the list of identified requirements:

- **Software functions**
  The software shall provide a template implementation of the binary tree approximation algorithm for mesh hp-refinement depicted in [1].
  
  **Type**: FUN
  **Verification**: TEST
  **Evidence**: Delivered package: example1, example2, example3, example4.

- **Multidimensionality**
  The software shall be able to manage binary trees whose nodes are constituted by generic items providing the capability to be divided in two; in particular, monodimensional (intervals) or bidimensional (triangle) items shall be implemented.
  
  **Type**: FUN
  **Verification**: TEST
  **Evidence**: Delivered package: example3.

- **Input and output mesh format**
  The software shall be able to read an input mesh from a file in .msh format1 and write the refined output mesh in a file in the same format.
  
  **Type**: FUN
  **Verification**: TEST
  **Evidence**: Delivered package: example1.

- **Configurability**
  The software shall be configurable for the following parameters:
  
  - Order of exactness of quadrature rule
  - Expression for the interpolating functions in both 1D and 2D cases
  - Number of iterations of the algorithm
  - Tolerance of the algorithm

1Possibility to use other formats shall be left open.
2.1. Requirements

Type: OPE
Verification: TEST
Evidence: Delivered package: example1, example2, example3, example4.
Configuration files: binary_tree.conf, interpolating_functions.conf, mesh_quadrature.conf, sandia_quadrature.conf.

- Design independence
The program/library shall not depend on definitions of external libraries for tasks/functions not specific to the external libraries themselves. It shall be possible to substitute an external library (e.g., the library providing mesh management) with an equivalent one without modifying the design of the program and/or portions of the code not directly interfacing the library.\(^2\)

Type: DES
Verification: INSP
Evidence: This document: 2.2.3, 4.5

- Configuration parameters
Parameters driving the software functioning shall not be hard-coded, but specified via command line parameters and/or configuration files in ASCII format.

Type: DES
Verification: TEST
Evidence: Delivered package: example1, example2, example3, example4.
Configuration files: binary_tree.conf, interpolating_functions.conf, mesh_quadrature.conf, sandia_quadrature.conf.

- Configuration parameters management
Management of configuration parameters shall be done using the GetPot external package.

Type: RES
Verification: INSP
Evidence: Source files: HelpFile.h, HelpFile.cpp (class Cfgfile).
Configuration files: binary_tree.conf, interpolating_functions.conf, mesh_quadrature.conf, sandia_quadrature.conf.

- Environment
The software shall run on Linux, and be written in C++.

Type: RES
Verification: INSP
Evidence: Delivered package. This document: 4.2

\(^2\)This means that external libraries shall be properly encapsulated to hide their details to the rest of the software.
2.2 Architectural Design

Description of the single classes in the code can be found in the documentation package generated by the DoxyGen tool from the source code. In this section, it will be illustrated the non-trivial interactions and relations between them, in order to show the design choices made and implemented in the software.

2.2.1 The main algorithm implementation

The algorithm is implemented by the class MeshRefiner. This is an abstract base class, providing the definition of methods that shall be implemented by the derived specialization classes, each of them implementing the methods according to the specific external library used for the low level management of the mesh. A specialization of the Refiner for the case of the libMesh library is implemented in the software. Both the base class and the derived class are template classes, in which the template parameter is a positive integer representing the dimensionality of the elements constituting the mesh.

The class provides methods to initialize the underlying library, load a mesh, and setting the algorithm parameters (max number of iterations, error threshold tolerance), while the crude work is made by a call to the method Refine. From a dynamic point of view, the sequence of calls to refine a mesh is quite simple, and is described by the diagram 2.2

1. Obtain a pointer to the selected mesh refiner object through the implementing library name;
2. Initialize the object;
3. Load in the object the mesh to be optimized;
4. Start the refining algorithm;
5. Save the resulting mesh.

The MeshRefiner object (and the functor object inside the MeshRefiner itself) is obtained through a generic mechanism implementing the Abstract Factory design pattern, as described in the following section.

![Diagram](image.png)

Figure 2.1: The MeshRefiner class
2.2. Architectural Design

2.2.2 The Abstract Factory pattern implementation

The whole software has been designed to safeguard the algorithm independence from implementation details that depend on user choices (e.g., the particular interpolating function to be used during the refinement, or the particular quadrature algorithm to be used for integration) and to allow future expansions or optimizations to be brought in without causing massive code (and, even worse, design) reworking. The key to obtain this has been identified in a variant of the Abstract Factory design pattern, whose implementation is shown in diagram 2.3.

![Abstract Factory diagram](image)

Figure 2.3: Abstract Factory diagram

The Factory class itself is implemented as a Singleton (no variation on the Singleton design pattern as defined in [3]), in order to ensure that builders registered at any time or at any execution point are not mixed up (see next section). Finally,
due to the presence of several factories in the software\footnote{refer to the DoxyGen documentation} in order to avoid extensive code replication, the \texttt{Factory} class is implemented as a template class, in order to facilitate maintainability and shorten development times.

2.2.3 The Plugin pattern implementation

The Abstract Factory pattern implementation described in the previous paragraph, which serves well in avoiding propagation of modifications to code in case of different options for implementation details, still requires at least recompilation and relink. This could have some drawback in case of distributed usage, or in case new versions of the same function is added to the framework (i.e., a new mesh managing library, or the debug version of the same library). What we want is to use it along with the previous one(s) specifying which one to use at runtime, by configuration. This is obtained exploiting the runtime library loading functions of the Linux operating system, using the Plugin pattern, whose implementation allows to attach a library to an executable at runtime, without the need to recompile and/or relink the main application. The building blocks of the pattern are shown in the component diagram

![Figure 2.4: The Plugin pattern](image)

The main application will load the library, that will comply to the specific interface mechanism, then within its initialization code will register itself to the proper factory. Provided plugin loading has been executed by the main before starting the actual operations, when the factories will be queried they will return the proper object, as registered by the plugin at load time. The dynamic runtime library loading is implemented by the \texttt{PluginLoader} and \texttt{Plugin} classes, the first acting as a container of plugins, and the second implementing the actual load/unload of the single library.

The sequence diagram\footnote{refer to the DoxyGen documentation} shows the process of runtime loading of plugins.
2.2. Architectural Design

The process runs as follows:

1. The main program creates a PluginLoader helper class, and specifies the libraries to be loaded at runtime (plugins), calling the Add method as many times as needed, once for each library.

2. The PluginLoader creates one Plugin object for each call, and stores them internally along with all other parameters needed to properly call the operating system service.

3. When all the libraries have been specified, the main program calls the Load method of the PluginLoader. This will cycle through the Plugin objects, and call the Open method for each of them.

4. For each plugin, the Open method, after all necessary checks, calls the dlopen operating system call, that instructs Linux to load the library and make it ready for execution, dynamically linking it to the main code.

5. As part of this function call, just at the beginning, the static function RegisterFunction, that must be provided within the library code, gets called, to carry out any needed initialization.

6. The RegisterFunction simply creates the builder objects that will, in turn, create the objects implementing the specific library tasks (e.g., MeshRefiner), and registers them inside the proper Factory. Being Factory objects singletons, the registration will go along all other similar registrations made by other libraries or from inside the main code.

7. At this point, the main code will be able to proceed using the library services through the Factory.
Note that interaction with the Factory goes along the Singleton pattern, accessing it with the **Instance()** method.

### 2.2.4 The *meta Factory* implementation

When combining the previously described patterns implementation (Factory + Plugin + Singleton) with the template implementation, an issue must be solved that can invalidate the whole approach: being the Factory+Singleton implemented as a template, its code is contained in a header file, which is included at compile time both in the main code (that, for instance, accesses the factory to get an object) and in the code of a library loaded as a plugin (that, for instance, accesses the factory to register a builder). This is unfortunate, because upon call of the **Instance()** method of the Singleton would result in creation of two singleton objects, one in the address space of the main, and one in the address space of the library if the call from the library is executed before the call from the main. To avoid the issue, all factories must be created in the main address space, ensuring that this creation is executed before any plugin is loaded. This is obtained through the definition of the **InstanceHolder** class, that acts as a simplified Factory of factories. The **Instance()** method of the Singleton Factories instead of directly creating an instance of the class, as in the *pure* Singleton pattern, tries to get it from the **InstanceHolder** object (itself a singleton factory). If the factory has not yet been registered, then the **InstanceHolder** fails, and allows the **Instance()** method to create a new object (as per the Singleton pattern) and store it in the **InstanceHolder** itself. The **InstanceHolder** singleton is statically created inside the refine_binary library, that is linked to the main code (whether as a static or as a dynamic library). This ensure that it is created before the **main()** gets invoked, and therefore before any plugin gets loaded. The glue to hold it together is constituted by the **FactoryBase** class, from which all Factories are derived. Factories inside the **InstanceHolder** are traced by pointers to the **FactoryBase**, as requested by the Factory pattern, of which the **InstanceHolder** is an instantiation. The diagrams 2.6 show graphically the portion of design reported in this section.

The diagram 2.7a shows the dynamics of Factory creation in the most common case, when builder for a given Factory are created by a library loaded at run time via the plugin mechanism, and used inside the main code:

1. The main program starts, and this gets the **InstanceHolder** simplified factory created.

2. At this point, via the plugin mechanism, the external library is loaded.

3. The library, from inside its **RegisterFunction**, tries to get the singleton instance of the factory, in order to register its `<key, builder>` association, via
the static `Instance()` method.

4. The method, goes to the `InstanceHolder`, using the `Factory` name as a key.

5. Since, at this point, the `Instance()` method of the specific factory has never been called before, the `InstanceHolder` has no registration for this factory, and raises an exception.

6. The exception is caught by the `Instance()` method, and it knows that has to create an instance of the factory.

7. Therefore, a new instance of the factory is created, registered inside the `InstanceHolder`, and returned to the original caller (the `RegisterFunction` of the library).

8. The `RegisterFunction` has now a valid factory pointer, and uses it to register its `<key, builder>` association.

9. When the main code needs a builder in the specific factory, it gets a handle to the factory singleton with the `Instance()` method.

10. This time, when the `Instance()` goes to the `InstanceHolder` class, the `factory_name` is found, no exception is raised, and the `Factory` pointer is directly returned to the caller.

11. The main uses the returned `Factory` pointer to get the object it needs, via invocation of the registered builder through the `create()` method.

Diagram 2.7b shows the dynamics of factory creation in an alternate case, when for the same factory some builders are registered inside the main code, and some other later on inside libraries loaded with the plugin method:
1. The main program starts, and this gets the `InstanceHolder` created.

2. In this case it is the main that invokes the `Instance()` method of the factory to register its `<key, builder>` pair.

3. The interaction between `Factory` and `InstanceHolder` goes as before, and at the end, the singleton `Factory` instance has been created and registered in the `InstanceHolder`.

4. The library is loaded via the plugin mechanism, and invokes the `Instance()` method of the `Factory` to register its own builder (that can be new, or overwrite the previous one).

5. The `Factory` instance has already been created, therefore it is retrieved in the `InstanceHolder` and returned to the library, that is able to register its builder.

6. The main code can get objects of both types, according to their name, accessing the `Factory` instance through the `create()` method.

### 2.2.5 Working with the mesh

The software hands-out the mesh representation and basic operations (creating the mesh, adding and deleting nodes, ...) to external libraries, that can change from one run to another. Also, there are several different operations to be carried out in the nodes when traversing the mesh. The problem of making the refining algorithm independent from the mesh representation and traversal, while at the same time allowing different operations to be executed on the nodes without duplicating
Figure 2.7: Iterator+Command pattern

or modifying the mesh navigation code, is solved with a pattern that combines elements of the Iterator (delegating mesh traversal to the object managing the mesh) and Command (passing to the executor the command to be applied to each node encapsulated in an object with a fixed syntax) patterns.

The MeshRefiner, as explained in 2.2.1, is in charge of implementing the main algorithm, while the actual implementation of the mesh is demanded to the concrete library implementation (in our software, the libMeshRefiner class, encapsulating the external libMesh library). Traversing the mesh is therefore demanded to concrete libraries imposing them the implementation of the IterateActiveNodes method. This method accepts in input the command to be executed, encapsulated in an object of type Functor. It is up to the IterateActiveNodes method to traverse the mesh and passing all nodes defining the mesh to the command in input, before returning to the caller.

2.2.6 The Strategy pattern usage

In several points of the software a choice must be made on what particular algorithm must be used to accomplish some specific task. In many cases, there are different versions of the algorithm that can be used; the typical case is the quadrature algorithm, used in the mesh refinement to compute the value of the integrals over the elements of the mesh. There is a choice over a certain number of quadrature algorithms, that will give different performances and, given a particular algorithm, there can be different libraries providing different implementations of the algorithm itself. For our purposes, the Gaussian quadrature rule has been considered the best choice for the 1D case [4], and two implementations of the Gaussian Quadrature are available, one provided by the libMesh library, the other provided by the Sandia library. In these cases, the object using the algorithm does not implement it, neither directly nor via inheritance. Rather, there is a separate object implementing the algorithm, providing the same interface of the method invoked in the parent object, which is built and passed to the parent during the parent construction (actually, but other algorithms can be easily added to the toolkit.
2. The Software

![Diagram of Quadrature rule Strategy Pattern](image)

Figure 2.8: Quadrature rule Strategy Pattern

this is passed via a **Factory** accessed by a **key** read from a specific configuration file). The diagram [2.8](image) shows how the Strategy Pattern applies to the quadrature rule implementation.

### 2.2.7 The Flyweight pattern usage

As explained in [1.4.2](section), the process of integral evaluation is based on passing through identification of nodes in a standard element, and then scaling the results to the actual node. This process has to be repeated for all nodes, but it turns out that the standard elements are always the same, and in a very small number. It would be a huge waste of memory and time if mesh nodes simply stored their own local copy of the proper standard element. To solve this problem, the Flyweight pattern has been used to design this portion of the code: standard elements are not directly created by the constructor of the elements; rather, they are constructed once and for all on their first access, via services of the proper Factory, and a pointer to the standard element is stored inside the normal elements. The diagram [2.9](image) shows the design of this portion of the code.

### 2.2.8 Collecting statistics on algorithm execution

During the refining process it can be addressed the problem of collecting information on the refining algorithm execution at all iterations. An example of this is tracing the error at every iteration of the refinement, or visualizing the mesh evolution through the iterations. The solution is found using the variadic template arguments feature provided by the C++: the **Refine** method of the **MeshRefiner** object, besides carrying out the main algorithm, accepts in input a variable list of objects that are nothing else than instances of the `function<return_type>` class

---

5 Or both of them, and possibly more...
provided by the STL library.

The method prototype reads:

\[
\text{template <typename... Args>}
\text{void Refine (std::size_t n_iter, double tol, Args... funcs);}\]

After any iteration of the algorithm, the `Refine()` method, executes the functions received in input, recursively parsing the variadic template arguments:

```
Listing 2.1: Parsing variadic arguments

\text{template <typename... Args>}
\text{void Execute (std::function<void()> f, Args... funcs)}
{\text{f();}}
\text{Execute (funcs...);}\};

\text{template <std::size_t dim>}
\text{template <typename... Args>}
\text{void MeshRefiner<dim>::Refine (std::size_t max_iter, double tol, Args... funcs)}
{\text{Execute (funcs...);}\text{size_t n_iter = 0;}\text{while (n_iter <= max_iter && total_error > tol)}{\text{[snip] // do the actual refinement iteration}}
\text{Execute (funcs...);}\text{++n_iter;}\};
```
2.2.9 Managing configuration files

As per requirement [2.1] GetPot has been used to manage configuration files and command line options. However, using of this library has a couple of drawbacks:

• as it will be reported among the **Known limitations** inside [4.3] come workaround needs to be used until a patched version of libmesh is available in order to avoid name clash of the GetPot functions with their internal version used inside libmesh itself. It is advisable to have these workarounds concentrate in a single place;

• some behavior of the library can change across versions: for instance, in case of a non existent input configuration file, the version provided with libMesh (which is derived from a previous release of the GetPot itself) throws an exception, while the official GetPot version silently ignores the error. It would be advisable having a stable and consolidated behavior as a baseline, in order to avoid to modify the software in case of changes in the GetPot package.

The problem has been solved through the Proxy pattern, creating a wrapper class `Cfgfile` that guarantees a nothrow constructor in any case, providing anyway a simple boolean test to use in if-statements to check if the file opening has been correctly performed, and overloading the GetPot methods used in the library in order to immediately return with the default value in case the constructor has not opened the file correctly, and simply passing parameters on to the matching GetPot method otherwise. Finally, for any other purpose, the `Cfgfile` class provides a conversion operator to retrieve from the proxy a copy of the underlying GetPot object, if needed.

The design of this part of the library is shown in diagram [2.10]

![Diagram](image-url)

**Figure 2.10: The Cfgfile proxy**
2.3 Coding details

This section contains a more detailed description of the main software objects implementing the algorithm depicted in [1].

The software is organized around several libraries:

- **refine_binary**: it is the main library implementing the algorithm from an abstract point of view. Here are defined the abstract base classes whose pure virtual methods coding is deferred to external plugins; it gives the interfaces for these inherited classes, maybe dependent on external libraries, which are in charge of the practical implementation of abstract methods.

- **library_bridges/mesh_bridge**: a dynamic library that acts as a wrapper for libMesh library, hiding to the main refine_binary the libMesh dependency; it gives an implementation of the abstract refiner, exploiting libMesh as mesh manager.

- **quadrature_rules/sandia_quadrature**: a dynamic library that exploits the SandiaRules library to give an implementation of optimized Gaussian quadrature rule.

- **quadrature_rules/mesh_quadrature**: a dynamic library that exploits the libMesh library to give an implementation of multiple quadrature rules on different kinds of geometries.

- **plugin_loader**: a dynamic library that gives tools to handle with the load/unload plugin process.

- **interpolating_functions**: a dynamic library giving implementation for multiple functor objects that can be used to lead the algorithm refinement process.

2.3.1 Quadrature Rules

The class **QuadratureRuleInterface<dim>** gives the abstract interface for objects storing the quadrature rules. A quadrature object is just characterized by its exactness order, which must be given to the object constructor. The rule implemented is instead deferred to the implementation of derived class, i.e. the **SandiaQuadrature** implements the Gaussian quadrature rule (see [1.4.1]), while the **LibmeshQuadrature** is configurable to any of the rules made available by libMesh. Once the **Quadrature** object has been created, its points and weights can be accessed through the proper method.

6The code in this section has been extracted and summarized from the library source code. The aim is to illustrate the ideas behind the implementation, for a detailed description of the code refer to the Doxygen documentation delivered with the software.

7Remember they are intended to be defined on the reference element $\hat{\Delta}$. 

29
2. The Software

Listing 2.2: Interface for a quadrature rule

```cpp
template <std::size_t dim>
class QuadratureRuleInterface
{
  public:
    /* Get the points */
    QuadPointVec<dim> GetPoints() const;
    /* Get the weights */
    QuadWeightVec GetWeights() const;
  protected:
    /* exactness order*/
    size_t _order;
};
```

The QuadratureFactory is responsible for giving the appropriate quadrature rule fitting the geometry in use.

2.3.2 Mapping process

The class `Map<dim>` gives the abstract interface for objects implementing the mapping process. This holds true for both singular maps like \[1.12\] and affine maps \[1.13\]. The class gives tools to evaluate the underlying map, its inverse and its Jacobian at some point \(x\).

Listing 2.3: Interface for a map

```cpp
template <std::size_t dim>
class Map
{
  public:
    /* Evaluate the map */
    Point<dim> Evaluate (const Point<dim>&) const;

    /* Evaluate the inverse map */
    Point<dim> ComputeInverse (const Point<dim>&) const;

    /* Evaluate the Jacobian*/
    double EvaluateJacobian (const Point<dim>&) const;
};
```

In the affine case optimizations can be done, exploiting the algebraic representation \[1.13\]. The `AffineMap<dim>` object stores its representative matrix and translation vector, exploiting the power of `Eigen` external library to perform the needed algebra.
2.3. Coding details

Listing 2.4: Attributes of an affine map

```cpp
template <std::size_t dim>
class AffineMap : public Map<dim>
{
    protected:
        /* Linear application matrix */
        MatrixType<dim> _mat;

        /* Representative of the inverse application */
        MatrixType<dim> _inverse;

        /* Translation vector */
        VecType<dim> _trasl;

        /* Constant Jacobian of the map */
        double _jacobian;
};
```

MatrixType and VecType are simply wrappers for appropriate Eigen types, so that the Eigen dependency is concentrated in these classes definition.

2.3.3 Finite Elements

The class AbstractFiniteElement<dim> gives the abstract interface for the representation of the space \( P_p(\Delta) \) of polynomial functions on the geometrical element \( \Delta \) of dimensionality \( \text{dim} \). The class endows the underlying geometrical domain with two main properties:

1. It gives the tools to compute the numerical integral of a function on its underlying domain

2. It gives the tools to construct the polynomial basis for the space \( P_p(\Delta) \)

As depicted in 1.4.3 both these two operations are defined in terms of the corresponding ones on the standard element \( \hat{\Delta} \), so the class stores an object of type StdFElement<dim>, together with the needed AffineMap<dim>, to map the output of standard elements operations to the current domain. The interface for finite elements finally reads:
2. The Software

Listing 2.5: Interface for a finite element

template <std::size_t dim, BasisType FeType>
class AbstractFElement
{
    public:
        /* Get the quadrature nodes */
        Geometry::QuadPointVec<dim> GetQuadPoints() const;

        /* Get the quadrature weights */
        Geometry::QuadWeightVec GetQuadWeights() const;

        /* Evaluate the i-th basis function at input point */
        double EvaluateBasisFunction(size_t, const Geometry::Point<dim>&) const;

    protected:
        /* The polynomial degree */
        size_t _p_level;

        /* The standard element */
        shared_ptr<StdFElementInterface<dim, FeType>> _ref_felement;

        /* The affine map from the standard element */
        unique_ptr<Geometry::AffineMap<dim>> _map;
};

The correct standard element and affine map are given by, respectively, the
StdFElementFactory and the AffineMapFactory.

The $p$-level of the element can be modified at runtime, and so the size of the
polynomial basis, while the type of polynomial basis is defined at compile time by
the template parameter FeType.

The operation of computing the numerical integral of a function $f$ on the element
$\Delta$ simply reduces to a scalar product between the vector of weights QuadWeightsVec
and the vector of $f$-evaluations over quadrature nodes QuadPointVec<dim>.

To get the quadrature data and the basis functions evaluations on the standard
element the StdFElement<dim> stores the objects of QuadratureRuleInterface<dim>
type and the singular map to the standard IperCube<dim> (e.g. standard square in
2D), where the basis is effectively defined (see §1.4.3 and stored as TensorialBasis
object). The TensorialBasisFactory ensures that the correct basis is assigned to
the standard ipercube.

---

8Note that in the 1D case the standard ipercube is effectively the standard interval, so there is no
need to pass through the singular map step.
Listing 2.6: Interface for the std finite element

```cpp
template <std::size_t dim, ElementType Type, BasisType FeType>
class StdFElement
{
public:
    /* Get the quadrature nodes */
    Geometry::QuadPointVec<dim> GetQuadPoints() const;

    /* Get the quadrature weights */
    Geometry::QuadWeightVec GetQuadWeights() const;

    /* Evaluate the i-th basis function at input point */
    double EvaluateBasisFunction (size_t, const Geometry::Point<dim>&) const;

protected:
    /* The quadrature rule */
    unique_ptr<QuadratureRuleInterface<dim>> _quadrature_rule;

    /* The standard ipercube */
    shared_ptr<StdFIPerCube<dim, FeType>> _std_cube;

    /* The possibly singular map from standard ipercube to standard geometry */
    unique_ptr<Geometry::Map<dim>> _ipercube_map;
};

template <size_t dim, BasisType FeType>
class StdFIPerCube
{
    /* Evaluate the i-th basis function at input point */
    double EvaluateBasisFunction (size_t, const Geometry::Point<dim>&) const;

protected:
    /* The polynomial basis*/
    unique_ptr<TensorialBasis<dim>> _basis;
};
```
2.3.4 Binary Elements

There’s only one step left to get the objects, composing the binary tree, the algorithm can handle with. The abstract interface for these objects is given by the class `AbstractBinaryElement`, which in addition to storing all the element information needed by the algorithm (e.g. the parameters $e_p$, $\tilde{e}_p$, $E$, $\tilde{E}_p$, $q$, $s$), it also saves a `Functor` object representing the function to be interpolated, and the `AbstractFiniteElement` the functor has to be interpolated on. So this class is able to compute the projected function and, through the tools illustrated in previous paragraph, also the interpolation error. Once the projection coefficients are computed and saved, the proper class method

```cpp
template <size_t dim, BasisType FeType>
class AbstractBinaryElement
{
    double Projection (const Geometry::Point<dim>&) const;
};
```

acts as a functor, returning the projected function value at input point.

We remember that at this level the classes illustrated are all abstract, since they depend on the underlying geometry, which is coded and managed by external libraries. So in order to have usable objects a concrete implementation of these abstract classes, both `AbstractBinaryElement` and `AbstractFiniteElement` among them, has to be given by external plugin (e.g. `mesh_refiner`).

Listing 2.7: Attributes of a binary element

```cpp
class BinaryNode
{
    protected:
        double _tilde_error, _E, _E_tilde, _q;
        BinaryNode* _s_element;
};
```

```cpp
template <std::size_t dim, BasisType FeType>
class AbstractBinaryElement : public BinaryNode
{
    protected:
        /* The interpolation error */
        double _projection_error;

        /* The functor whose projection error has to be computed */
        FunctionPtr<dim> _f;

        /* The finite element the function _f is projected on */
```
2.3. Coding details

```cpp

unique_ptr<AbstractFElement<dim, FeType>> _f_element;

private:
    /* The vector of projection coefficients */
    CoeffVector _coeff;
};

2.3.5 The Algorithm

As anticipated in 2.2.1, the algorithm 1.2 is implemented at an abstract level in
the class MeshRefiner through the Refine method. It operates on binary nodes of
abstract type BinaryNode, so that the execution is independent on how the mesh
is stored, and so on how the binary nodes are implemented. These operations are
defered to inherited classes coded in external plugin, and an example of imple-
mentation is given by the library_bridges/mesh_bridge dynamic library, which
exploits libMesh library to manage the mesh object. Appropriate methods are given
to import/export the mesh from file, these possibly dependent on the external plugin
implementation too.

Listing 2.8: Interface for mesh refiner

template <size_t dim>
class MeshRefiner
{
public:
    /* Import mesh */
    void LoadMesh (std::string);

    /* Export mesh */
    void ExportMesh (std::string) const;

    /* Apply the algorithm */
    void Refine (size_t n_iter);

private:
    /* Perform the p refinement along the parents of input
    node */
    void ClimbUp (BinaryNode*);
};

The MeshRefiner class stores the functor representing the function to be approx-
imated, and an object of type DimensionedGodGather, which acts like a ghost node
father of the $\mathcal{R}$ element; this object is in charge of the bisection operation on the
$s(\mathcal{R})$ element, and of the trim operation 1.3. Moreover the DimensionedGodGather
is able to handle with a starting mesh with more than one element, as described in 1.2.

Once the bisection on the $s(R)$ leaf has been performed, the `MeshRefiner` updates the nodes, whose parameters have changed as a result of the addition of the just created leaves, through the `ClimbUp()` method.

Listing 2.9: Attributes of the mesh refiner

```cpp
template <size_t dim>
class MeshRefiner {
    protected:
        /* The function to be approximated */
        FunctionPtr<dim> _objective_function;

        /* The ghost node */
        DimensionedGodFather<dim> _godfather;
};
```

Listing 2.10: The ghost node implementation

```cpp
template <size_t dim>
class DimensionedGodFather {
    public:
        /* Refine the next leaf to be bisected returning the father of the new just created leaves */
        BinaryNode* MakeBisection();

        /* Perform the trim operation on the binary tree */
        void SelectActiveNodes();

    protected:
        /* Binary elements of the initial mesh. Every element could generate a binary tree */
        std::list<DimensionedNode<dim>>*_elements;
};
```

So finally the coded algorithm simply reads:

Listing 2.11: The refinement algorithm implementation

```cpp
template <size_t dim>
void MeshRefiner<dim>::Refine (size_t max_iter) {
    size_t n_iter = 0;
```

\[\text{Here the implementation of the more general version of the algorithm, described in 1.2. An implementation of its simpler one, as depicted in 1.1, is available too}\]
while (n_iter < max_iter && total_error > tol) {
    /* h-refine the s(R) node */
    BinaryNode* daddy = this->godfather.MakeBisection();

    /* Update the tree */
    ClimbUp (daddy);

    /* Make the optimal trim */
    (this->godfather).SelectActiveNodes();

    /* Compute global approximation error */
    total_error = this->GlobalError();

    ++n_iter;
}

2.4 Design Justification

As stated in several points in the previous sections, flexibility, reusability and ease of maintenance have been key drivers for the design and implementation of the software, leading to the adoption of the design patterns described in the logical architecture. This section contains the rationale behind some important choices made, that have been driven by other factors and considerations.

2.4.1 Mesh library selection

Once the decision has been taken not to implement directly the mesh management code, but instead rely on an external off-the-shelf library, the problem of which library select had to be faced. Several alternatives were available. Therefore, in order to make the choice the most objective possible, each of them has been evaluated against a set of feature, specifying for each library the level of support for each feature (VERY LOW, LOW, FAIR, GOOD, VERY GOOD), and ranking all features according to the importance they had for the project (LESS IMPORTANT, NICE TO HAVE, IMPORTANT, VERY IMPORTANT, MANDATORY). At the end, the library scoring the highest weighted score has been determined (that is, libMesh), and has been selected for being integrated in the library. Remains the fact that other libraries can be integrated in future developments (refer to §4.5 for the details).

The following libraries candidate for selection have been evaluated:
2. The Software

- libMesh\textsuperscript{10}
- deal II\textsuperscript{11}
- p4est\textsuperscript{12}
- MFEM\textsuperscript{13}
- ALBERTA\textsuperscript{14}
- hpGEM\textsuperscript{15}
- AptoFEM\textsuperscript{16}

The libraries evaluated, and their scores against each feature, together with its importance, are shown in table 2.2.

<table>
<thead>
<tr>
<th>criterion</th>
<th>Imp</th>
<th>libMesh</th>
<th>deal II</th>
<th>p4est</th>
<th>MFEM</th>
<th>ALBERTA</th>
<th>hpGEM</th>
<th>AptoFEM</th>
</tr>
</thead>
<tbody>
<tr>
<td>1D and multidimensional support</td>
<td>M</td>
<td>VG</td>
<td>G</td>
<td>L</td>
<td>VG</td>
<td>VG</td>
<td>VG</td>
<td>L</td>
</tr>
<tr>
<td>hp high order refinement</td>
<td>M</td>
<td>VG</td>
<td>VG</td>
<td>VG</td>
<td>VG</td>
<td>VG</td>
<td>VG</td>
<td>VG</td>
</tr>
<tr>
<td>tree inner structure</td>
<td>M</td>
<td>VG</td>
<td>VG</td>
<td>VG</td>
<td>VG</td>
<td>VG</td>
<td>VG</td>
<td>VG</td>
</tr>
<tr>
<td>c++ compatibility</td>
<td>VI</td>
<td>VG</td>
<td>VG</td>
<td>G</td>
<td>VG</td>
<td>G</td>
<td>VG</td>
<td>L</td>
</tr>
<tr>
<td>triangular mesh with hanging nodes support</td>
<td>VI</td>
<td>VG</td>
<td>L</td>
<td>VL</td>
<td>VL</td>
<td>L</td>
<td>VG</td>
<td>F</td>
</tr>
<tr>
<td>algorithm independence from geometry</td>
<td>I</td>
<td>VG</td>
<td>VG</td>
<td>VL</td>
<td>G</td>
<td>L</td>
<td>VG</td>
<td>L</td>
</tr>
<tr>
<td>ease of learning</td>
<td>N</td>
<td>VG</td>
<td>VG</td>
<td>G</td>
<td>F</td>
<td>L</td>
<td>F</td>
<td>VL</td>
</tr>
<tr>
<td>support to bisection implementation</td>
<td>VI</td>
<td>G</td>
<td>L</td>
<td>VL</td>
<td>L</td>
<td>F</td>
<td>G</td>
<td>VL</td>
</tr>
<tr>
<td>well documented</td>
<td>N</td>
<td>VG</td>
<td>VG</td>
<td>G</td>
<td>F</td>
<td>F</td>
<td>F</td>
<td>VG</td>
</tr>
<tr>
<td>widespread usage</td>
<td>L</td>
<td>G</td>
<td>VG</td>
<td>VG</td>
<td>G</td>
<td>F</td>
<td>F</td>
<td>G</td>
</tr>
</tbody>
</table>

weighted score

<table>
<thead>
<tr>
<th></th>
<th>libMesh</th>
<th>deal II</th>
<th>p4est</th>
<th>MFEM</th>
<th>ALBERTA</th>
<th>hpGEM</th>
<th>AptoFEM</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>246</td>
<td>462</td>
<td>530</td>
<td>448</td>
<td>438</td>
<td>540</td>
<td>514</td>
</tr>
</tbody>
</table>

This method has ensured the maximum level of objectivity in the selection process. Still, both the single library scores and the feature ranking remain kind of subjective, and the selection can be challenged on the basis of a different ranking of the features and/or different evaluation of the level of support of some feature by some library. This has led to the design choice to allow with the minimum possible effort the adoption of a new library without even recompiling the source code of the library (refer to 2.2.3).

2.4.2 Template vs polymorphism

The algorithm implementation in the library is heavily based on templates. The usual trade-off between templates and polymorphism has been resolved in favor of

\[\text{http://litmsh.github.io}\]
\[\text{http://www.dealii.org}\]
\[\text{http://www.p4est.org}\]
\[\text{http://mfem.org}\]
\[\text{http://www.alberta-fem.de/index.html}\]
\[\text{http://hpgeom.org}\]
\[\text{https://www.maths.nottingham.ac.uk/personal/ph/Site/Software.html}\]
the first, chiefly because of performance reasons. In fact, even though no explicit performance requirements have been imposed on the software, yet avoiding long execution times, in particular when the mesh size starts to grow, was still identified as a key driver for the design. The first major drawback of template usage, i.e., code bloat, has been evaluated less impacting, both because optimizing compilers can reduce it on their own, and because it is not very probable that programs will use more than a couple of instantiation of the templatized classes. The other major drawback, i.e., the need to know the exact version of the concrete object at compile time, reducing to the dimensionality of the mesh, is not an issue as well, since the dimensionality of the mesh is fixed and stated in the specifications.

Finally, when it comes down to the low level implementation of basic mesh building blocks, almost all evaluated libraries adopted the same approach. On the other end, on the performance side, adoption of templates allowed some interesting and efficient solutions based on the template programming technique, as in the following methods/functions:

- The method `UpdateSize()` of class `TensorialBasis` uses this technique to tensorize the basis functions\(^{17}\).
- The functions `TensorizePoints()` and `TensorizeWeights()` use this technique for tensorization of quadrature nodes\(^{18}\).
- The function `Power` uses this technique to exploit knowledge of the template dimension and of the fact that exponents are positive integers, and fast and efficiently calculate at compile time the value of the powers\(^ {19} \).

\(^{17}\)File `refine_binary/include/Basis.h`  
\(^{18}\)File `quadrature_rules/sandia_quadrature/include/SandiaQuadrature.h`  
\(^{19}\)File `refine_binary/include/BinaryTreeHelper.h`
3 Numerical Tests

In order to validate the algorithm and the correctness of its implementation in the library, numerical tests have been made on 1D and 2D problems.

The results of these simulations have then been compared with the expected results already shown in [1.3] in order to check that the refinement process performs as predicted by the theoretical analysis.

3.1 1D mesh

For the 1D case, the (0, 1) interval has been considered, and for the refinement process the $L^2$-norm of the interpolation error has been considered as the $e_p(\Delta)$ quantity.

Several different functions have been tested, taking into account that:

- a polynomial function is expected to be interpolated exactly for a sufficiently high complexity of the algorithm, which should generate a single block with high order basis degree;
- an analytic function is expected to have an approximation error which decreases following the profile given by [1.20];
- a piecewise analytic function is expected to have an approximation error which decreases following the profile given by [1.23].

For each test case, three plots will be shown, representing the obtained profile of the interpolation error against increasing values of complexity of the algorithm, the resulting refined mesh, the comparison between the analytic function objective of the algorithm and its interpolation on the aforementioned refined mesh.

3.1.1 Polynomial $f(x)$

For the case of a polynomial function, given a sufficiently high complexity parameter, since the projection coincides with the function itself, the algorithm is expected

---

1 The (0, 1) interval on the x-axis; the p-levels on the y-axis; different color corresponds to different element.
2 For all tests the tolerance on the error have been fixed to $\epsilon = 10^{-14}$.
3 Plots obtained from the example2 (properly configured) output.
3. Numerical Tests

(a) Error profile
(b) Interpolated function
(c) Refined mesh $p$-levels

Figure 3.1: $f(x) = (2x - 1)^{40}$

...to approximate exactly $f(x)$. In order to verify this, a numerical test has been executed on the function $f(x) = (2x - 1)^{40}$. Results of this test are shown in figure 3.1.

It can be seen that the algorithm, as expected, generates a single block mesh with the highest possible $p$-level, with a final interpolation error which, with complexity $N = 40$ or higher, is of the order of computer numerical representation errors.

3.1.2 Piecewise polynomial $f(x)$

As in the previous case, given a sufficiently high number of iterations for the algorithm, a piecewise polynomial function is expected to be exactly approximated, too, if the discontinuity points correspond to mesh nodes.

For this purpose the function $f(x) = x^2 \mathcal{H}(x - \frac{1}{2})$, represented in figure 3.2b together with its interpolated one, has been tested; results are shown in figure 3.2.

Again, the result corresponds to the expected one, with the algorithm stopping after 3 iterations, since the interpolation error is below the tolerance value.

---

*Plots obtained from the example2 (properly configured) output.*
3.1.3 Analytic $f(x)$

An analytic function should have an error decreasing following the estimate $O(20)$; so what is expected is an approximately linear profile in semilogarithmic scale. For this purpose the function

$$f(x) = \frac{e^{\frac{b}{\mu}x} - 1}{e^{\frac{b}{\mu}} - 1}$$

has been tested, which is the analytic solution of the $1D$ advection diffusion problem (see [4]):

$$\begin{cases} 
-\mu u'' + bu' = 0 & 0 < x < 1 \\
 u(0) = 0, u(1) = 1 
\end{cases}$$

with $\mu > 0$, $b > 0$. In figure 3.3 results for the case $\frac{b}{\mu} = 100$ are shown.

It can be seen that the test results fulfill the theoretical prediction.
3. Numerical Tests

(a) Error profile

(b) Interpolated function

(c) Refined mesh $p$-levels

Figure 3.3: $f(x) = \frac{e^{\mu x} - 1}{e^\mu - 1}$

3.1.4 Piecewise analytic $f(x)^6$

The last monodimensional test case is the one of a piecewise analytic function. In this case the expected profile for the interpolation error is the one given by the estimate \[1.23\] with a mesh finer and low $p$-refined near the singularities, while coarser, with increasing $p$-levels, getting away from them. As an example of this case the function $f(x) = \sqrt{x}$ has been tested; test results are shown in figure 3.4.

As it can be seen in 3.4a, the error profile fits very well the theoretical one, so the most important result, which confirms the optimality of the algorithm, has been verified. However the obtained mesh is pretty much the one expected, except for the element nearest the singularity point $x = 0$, which shows a huge $p$-level refinement. To deeper investigate this case, the obtained result has been compared with a manually refined mesh, with same complexity, but different allocation of $p$-levels on the same $h$-refined partition; the manual imposed $p$-levels should better follow the distribution expected from theory, and are shown in 3.5a.

\(^6\)Plots obtained from the example2 (properly configured) and example4 outputs.
Comparing the result with the one, it can be seen that the error is increased; in particular, calling $\mathcal{E}_m$ the approximation error on the manual refined mesh:

$$\mathcal{E} = 1.3672 \cdot 10^{-5}$$
$$\mathcal{E}_m = 3.7215 \cdot 10^{-4}$$
$$\mathcal{E} - \mathcal{E}_m = -3.5848 \cdot 10^{-4}$$

So it seems that the algorithm has led to an optimal result. A deeper investigation of
3. Numerical Tests

this case has been left to future developments, anyway we may raise an hypothesis, that is: the observed discrepancy in 3.4c w.r.t. the theoretical prediction may be due to the fact that the singularity point coincides with a mesh node.

To check this, a second test have been made on the function \( f(x) = \sqrt{|x - \frac{1}{3}|} \), where the singularity, now in \( x = \frac{1}{3} \), has been moved to a point that can never be a mesh node obtained by binary refinement. The results are illustrated in figure 3.6 and show a great accordance with the theoretical predictions.

Figure 3.6: \( f(x) = \sqrt{|x - \frac{1}{3}|} \)
3.2 2D mesh

To test the algorithm on a 2D domain, it has been considered the $[0, 1] \times [0, 1]$ square discretized using triangular elements; the criterion to perform $h$-refinement on these elements is the one already depicted in 1.3. The starting mesh the algorithm has been applied on is represented in 3.7.

![Starting mesh](image)

Figure 3.7: Starting mesh

As for the monodimensional case, we have firstly checked that a polynomial function is exactly approximated; the considered function is:

$$(2x - 1)^{10} (2y - 1)^{10}$$

and the results, illustrated in figure 3.8 confirm the hypothesis.

![Error profile and refined mesh](image)

Figure 3.8: $f(x) = (2x - 1)^{10} (2y - 1)^{10}$

Then a more challenging test has been applied to the function:

$$\left| \sqrt{x^2 + y^2} - \frac{1}{3} \right|^\frac{1}{2}$$

This function tries to replicate, in a bidimensional framework, the one of 3.1.4 that in the monodimensional case has given a great feedback. For this case, the expected

\footnote{Plots obtained from the example3 (properly configured) output.}
profile for the error is the one given by the estimate \(1.24\) results are shown in \(3.9\)

![Error vs Complexity](image)

(a) Error profile

![Refined mesh, the singularity in dotted line](image)

(b) Refined mesh, the singularity in dotted line

Figure 3.9: \(f(x) = \left| \sqrt{x^2 + y^2} - \frac{1}{3} \right|^\frac{\bar{h}}{2}\)

The error profile fits very well the theoretical one, and the mesh is finer near the singularity.\(^8\) Anyway, the bidimensional case needs to be investigated more deeply, to test the behaviour of the algorithm with different functions other than the ones proposed, and different meshes over more complex geometries, too. These cases shall be investigated first from the theoretical point of view, then from the numerical and implementative one. In order to do this, also the input/output section of the software shall be improved. These tests for the multidimensional case are left for future development.

---

\(^8\)p levels are reported in \(p\_levels\_[n]\) file, manually added to the 3.9b plot for representation purposes
4 User manual

4.1 Software description

The library consists of a series of shared objects and a set of examples to show how to use the library, delivered as a zipfile package containing source code and whatever necessary (e.g., Makefiles) to build the binary code from the source. The package can be downloaded from the GitHub repository, at the following link:

https://github.com/soblinsky/binary_adapt

For both libraries and binaries, two versions are created during the installation package:

- a normal, optimized version
- a debug version, suitable to allow symbolic debugging with standard Linux tools (e.g., gdb)

The style adopted for writing the C++ code has been taken from the GNU Coding Style and has been enforced using the astyle tool. The details of the formatting rules are distributed with the package, and can be found in the file binary_adapt.astylerc.

4.2 Setup and initialization

Requirements

Base requirements to build the package:

- GNU-compatible Make or gmake
- POSIX-standard shell
- A 2014 ISO C++ standard compliant GNU compiler

https://gcc.gnu.org/wiki/CppConventions

Contents of this section are also available in the software installation package inside the README file.
4. User manual

Dependencies

Before the software can be built, the following packages must be installed:

- **Eigen**: this library provides the tools to compute linear algebra operations, needed to implement the mapping process from the reference element to the generic element;

- **GetPot**: this library is needed for the parsing of the configuration files;

- **GTest**: library needed to build the testing framework and automatize the check of the correctness of the installation;

- **MuParser**: this library provides runtime parsing of mathematical expression from string; it is useful to arbitrarily set the function to be interpolated by the algorithm without recompiling the code;

- **LibMesh**: library for the numerical simulation of partial differential equations, it is needed to manage the mesh object and the refinement process. It is also exploited to implement quadrature rules for any handled geometry.

Note: the LibMesh library must have been built without MPI support (refer to LibMesh installation instructions: `configure --disable-mpi`).

- **JacobiPolynomial**: this library provides the evaluation of the Jacobi polynomials needed to build the polynomial basis over mesh elements;

- **SandiaRules**: library needed for the implementation of optimized Gaussian quadrature rules for any exactness order required.

Note: these two ones are libraries released by source, so they have been included in `binary_adapt` release, together with suggested instructions to compile them. It is understood that the user can provide his own installation of these libraries.

Pre-installation configuration

Download the distribution archive and unzip it somewhere, or clone the repository. Edit the `Makefile.inc` and set the environment variables therein to the correct paths for the dependent libraries.

---

3version 3.3.4 or higher - http://eigen.tuxfamily.org
4version 1.1.17 or higher - http://getpot.sourceforge.net
5https://github.com/google/googletest
6http://beldoforun.de/article.php?a=muparserx
7version 1.1.0 or higher - http://libmesh.github.io
8http://people.sc.fsu.edu/~jburkardt/cpp_src/jacobi_polynomial/jacobi_polynomial.html
4.3 Nominal operations

Installation

Do:

1. make
2. make check (optional, runs the test programs)
3. make install

The make install step will create the libraries in the lib folder. Modify the installation destination by setting the InstallDir environment variable in Makefile.inc.

To generate the documentation do:

- make doc

To generate the example binaries do:

- make examples

4.3 Nominal operations

After installation, the software is ready to use. The user can modify its internal working parameters editing the following configuration files:

- binary_tree.conf: configure the parameters of the algorithm;
- quadrature_rules/mesh_quadrature/include/mesh_quadrature.conf: configure the libmesh quadrature rule;
- quadrature_rules/sandia_quadrature/include/sandia_quadrature.conf: configure the sandia quadrature rule;
- interpolating_functions/include/interpolating_functions.conf: configure the muparser expression (remember to set the muparser functor in binary_tree.conf in order to use muparser framework).

Pass as main parameter your own configuration file in alternative to the default one binary_tree.conf; other configuration files are not replaceable.

Running the examples

The software is distributed with four examples. They are created in the folders example/example[n]/bin, with [n] ranging from 1 to 4, in both normal and debug version, and can be launched with the commands:

- ./Example_run.sh [n]
- ./Example_debug.sh [n]

Contents of this section are also available in the software installation package inside the README file.
Known limitations

1. libMesh redefines GetPot classes without including them in proper namespace, so the usage of both GetPot and libMesh library is not possible at the moment because of this conflict which causes unpredictable behaviour of the program. Bug has been reported to libMesh developers, and private communication started to fix the problem: at the moment a patch is under development. The patched libMesh will allow the user to configure libMesh with --with-getpot-namepace=libMesh flag, so that any conflict will be avoided. Until the patch release the libMesh version of GetPot has to be used also by plugins which do not need any other libMesh object.

2. At the end of each example a "Memory leak detected!" warning is printed at standard output. This is due to the libMesh mode of operation, which assumes that every libMesh instruction is called inside a LibMeshInit block. The correctness of the implementation, and so the absence of any memory leakage, has been verified via valgrind tool. To make the check do:

./Example_valgrind.sh [n]

Bug reports, feature requests

In case there are any bugs, or to give suggestions about the add of new features in the library, the following email address can be notified:

simone.carriero92@gmail.com

4.4 Error conditions

The following paragraphs report the main errors detected by the software during execution, how they are signaled, and what to do to resolve the problem (if possible). After resolution, the program can be restarted (after recompiling it, if needed).

Configuration errors

1. Error: Empty interpolating function
   Description: The tag mu_parser_expr is empty
   Signaling method: Exception thrown: “Error reading the configuration file in ParserFunctor: empty string expression”
   Recovery action: Fill the value of tag mu_parser_expr in file interpolating_functions.conf.

2. Error: Bad interpolating function
   Description: Expression in the tag mu_parser_expr has a wrong syntax
4.4. Error conditions

Signaling method: Exception thrown: “Content of muParser exception

Recovery action: Fix the syntax of tag mu_parser_expr in file interpolating_functions.conf.

3. Error: Unknown mesh refiner
Description: No builder for the configured mesh refiner library is found
Signaling method: Exception thrown: “Identifier mesh_library is not stored in the factory”
Recovery action: Correct the value of tag mesh_library in file binary_tree.conf.
Ensure that the library specified by the tag is loaded via the plugin mechanism.
Ensure that the library specified by the tag registers itself with the tag mesh_library in the factory.

4. Error: Unknown quadrature rule
Description: No builder for the configured quadrature rule is found
Signaling method: Exception thrown: “Identifier quad_library is not stored in the factory”
Recovery action: Correct the value of tag quad_library in sections [/ipercube] and/or [/triangle] in file binary_tree.conf.
Ensure that the library specified by the tag is loaded via the plugin mechanism.
Ensure that the library specified by the tag registers itself with the tag quad_library in the factory.

5. Error: Unknown function library
Description: No builder for the configured function library is found
Signaling method: Exception thrown: “Identifier func_library is not stored in the factory”
Recovery action: Correct value of tag func_library in file binary_tree.conf.
Ensure that the library specified by the tag is loaded via the plugin mechanism.
Ensure that the library specified by the tag registers itself with the tag func_library in the factory.

Execution errors

1. Error: Missing library
Description: A runtime loaded library is not found by the Plugin manager
Signaling method: Exception thrown: “Cannot load <library name>”
Recovery action: Correct value of tags func_library, mesh_library and quad_library in file binary_tree.conf.
4. User manual

Ensure that the libraries specified by the tags exist and have read and execute rights granted.

2. **Error**: Cannot open input mesh  
   **Description**: A wrong input mesh filename has been specified  
   **Signaling method**: (by libmesh library) Message Logged “Stream is bad! Perhaps the file does not exist?” and Exception thrown: “Error in libMesh internal logic"  
   **Recovery action**: Correct value of tag `input_mesh` in file `binary_tree.conf`. Ensure that the mesh file specified by the tag exists and has read rights granted.

**Coding errors**

1. **Error**: Mesh not initialized  
   **Description**: A mesh is being used but it has not been initialized  
   **Signaling method**: Exception thrown: “Trying to use uninitialized mesh”  
   **Recovery action**: Ensure that the `Init()` method of the `MeshRefiner` object is called before using it.

2. **Error**: Wrong Point initialization  
   **Description**: There is a mismatch between the dimension of the `Point<dim>` object and the number of coordinate initializers provided in the constructor  
   **Signaling method**: Exception thrown: “Invalid number of elements in Geometry::Point construction”  
   **Recovery action**: Ensure that the number of coordinates passed to the `Point<dim>` constructor has exactly `dim` elements.

3. **Error**: Incorrect `GetMesh` usage  
   **Description**: The library is not being used as a component of an external program using libMesh (see 4.5), but the `GetMesh` method of the concrete class `LibmeshRefiner<dim>` is being called, which is not available in this use case  
   **Signaling method**: Exception thrown: “The mesh can’t be exported outside the class that owns the LibMeshInit communicator”  
   **Recovery action**: Modify the code to avoid calling the `GetMesh` method of the library. The mesh is available in the function or object that passed the mesh to the library with a previous `SetMesh` call.

4. **Error**: Incorrect PluginLoader usage  
   **Description**: The `PluginLoader` is being used to load libraries at runtime, and libraries are added after the `Load` method has been invoked. Libraries added after the call to `Load` will be ignored  
   **Signaling method**: Message logged: “Warning: adding a plugin, but the Load() has already been called"
**Recovery action:** Modify the code to ensure that the `Load()` method is called after all libraries have been added.

### 4.5 Non nominal operations

**How to use the library inside an external program**

The library wrapping libMesh functions (`libmesh_bridge.so`) can be used to add the binary tree approximation algorithm to an existing program that is already using libMesh on its own. To do this, the following steps must be followed:

1. Create the mesh inside the program, and save its pointer in an object of type `std::shared_ptr<libMesh::MeshBase>`.  
2. Call the `BinaryTree::Init` function, passing as input the filename of the configuration file if different from the default one.  
3. Create an object of the class `LibmeshRefiner<dim>`, with `dim` set to the proper dimensionality.  
4. Pass the mesh pointer to the refiner object calling the `SetMesh` method of the refiner.  
5. Activate the refining calling the `Refine` method of the refiner.  
6. Retrieve the pointer to the optimized mesh calling the `GetMesh` method of the refiner.

**How to use an alternate mesh managing library**

If a library different from libMesh has to be used to provide the refiner with mesh representation and management capabilities, the following steps must be followed:

1. Write a class implementing the interface `BinaryTree::MeshRefiner<dim>`. In particular, ensure that the wrapped library mesh nodes are in some way convertible back and forth to objects of type `BinaryTree::DimensionedNode<dim>` (this can require different levels of effort and insights in the library internals, according to the way the library itself implements the mesh tree structure)  
2. Write the function that will be called upon runtime load of the library by the operating system. This function shall have the following prototype:

```cpp
__attribute__((constructor))
static void RegisterFunction()
```
and in its body register a builder in the mesh refiner factories of the supported dimensions, e.g.:

Listing 4.1: Register different mesh refiner

```cpp
auto& r_one_d_factory (  
    BinaryTree::MeshRefinerFactory<1>::Instance());
auto& r_two_d_factory (  
    BinaryTree::MeshRefinerFactory<2>::Instance());

r_one_d_factory.add (  
    "new_lib_name",
    &Helpers::Builders <  
    NewOneDLibRefiner,  
    BinaryTree::MeshRefiner<1>::BuildObject);

r_two_d_factory.add (  
    "new_lib_name",
    &Helpers::Builders <  
    NewTwoDLibRefiner,  
    BinaryTree::MeshRefiner<2>::BuildObject);
```

3. Compile the code and generate a runtime loadable library (using the flags `-fPIC -rdynamic`)

4. Set the tag `mesh_library` in the `binary_tree.conf` configuration file to the name of the wrapper library (without the leading `lib` prefix and without the trailing `.so` suffix), and the tag `mesh_refiner` to one of the keys registered into the factory, according to the selected refiner for usage inside the algorithm.

The `library Bridges/mesh_refiner` package, wrapping libMesh library, provided with the software can be taken as a template for construction of the new library for include files, source files and `Makefile`.

**How to use an alternate quadrature rule**

If a library different from libMesh or SandiaRules has to be used to provide the refiner with quadrature rules implementation, the following steps must be followed:

1. Write a class implementing the interface `QuadratureRuleInterface<dim>`

2. Write the function that will be called upon runtime load of the library by the operating system. This function shall have the following prototype:

```cpp
__attribute__((constructor))
static void RegisterFunction()
```
and in its body register a builder in the quadrature factories of the supported dimensions, e.g.:

Listing 4.2: Registered different quadrature rule

```cpp
auto& q_one_d_factory (  
    Geometry::QuadratureFactory<1>::Instance());
auto& q_two_d_factory (  
    Geometry::QuadratureFactory<2>::Instance());

q_one_d_factory.add (  
    Geometry::IntervalType,  
    &Helpers::Builders <  
    NewIntervalQuadrature,  
    Geometry::QuadratureRuleInterface<1>  
                  ::BuildObject);  

q_two_d_factory.add (  
    Geometry::TriangleType,  
    &Helpers::Builders <  
    NewTriangleQuadrature,  
    Geometry::QuadratureRuleInterface<2>  
                  ::BuildObject);  
```

Note that in this case the key to the factory is constituted by one of the node types available in the main program (Interval, Square, Triangle), and not by the name of the library.

3. Compile the code and generate a runtime loadable library (using the flags `-fPIC -rdynamic`)

4. Set the tag `quad_library` in the `binary_tree.conf` configuration file to the name of the wrapper library (without the leading `lib` prefix and without the trailing `.so` suffix) in the sections for the supported element types (`[./ipercube]` and/or `[./triangle]`).

The `quadrature_rules/sandia_quadrature` package provided with the software can be taken as a template for construction of the new library for include files, source files and `Makefile`.

**How to use an alternate set of interpolating functions**

If a library different from the one provided in the package has to be used to provide the refiner with the objective function of the algorithm, the procedure is the same as for the refiner, taking care of properly setting the value of the configuration flags.
4. User manual

The *interpolating_functions* package provided with the software can be taken as a template for construction of the new library for include files, source files and *Makefile*.

How to use library components outside the refinement problem

The following classes/packages are used inside the refiner for the algorithm implementation, but have been designed with reusability in mind, and can be used on their own or extended with little effort (if deemed useful, of course):

1. Quadratures
2. Linear Algebra
3. Finite Elements
4. Logfile
5. Cfgfile
6. PluginLoader

In order to use them, just include the proper header files in the code, and link (or load via the PluginLoader) the proper library. Details on single classes usage can be found in the Doxygen documentation available in the package and buildable with the command `make doc`.
Conclusions

In this project, we have approached the problem of approximating a function exploiting mesh adaptation to solve it in a smart and efficient way. This kind of problem, and our approach in solving it as well, can be generalized and used as a starting point for the study of a wide class of practical PDE problems, whose numerical solution often needs mesh adaptivity to find a sufficiently accurate result in reasonable computational time. In particular, there are lots of problems, e.g. the high order Discontinuous Galerkin solution of PDE equations, in which a substantial improvement on the solution accuracy can be obtained by \( p \)-refinement of mesh elements, that is, by increasing even to huge values the degree of polynomial basis built on the element, other than \( h \)-refining it.

Our work provides a tool that not only exploits \( hp \)-adaptivity to speed up the simulation, but it does it in a near-optimal way, so that, fixed the complexity of the to be obtained mesh, it overcomes in terms of performances most of other possible combinations of the \( h \) and \( p \) version of the refinement.

The theoretical result for the \( hp \)-adaptivity algorithm, given by [1], has been taken as starting point of our work; its theoretical approach has been integrated with the practical tools necessary to apply it to concrete problems.

In [2] we have made the needed generalizations to include common working meshes in the range of applicability of the algorithm, then we have built the mathematical structure to compute the quantities the algorithm works with.

Then in [3] we have given an implementation of the theoretical framework depicted, programming an open source C++ suite that allows to exploit the algorithm to \( hp \)-refine 1D meshes and 2D triangular ones, too; the general approach kept in coding has led us to a versatile and easy extendable toolkit, that can be expanded in future developments or adapted by external users to their working frameworks. A detailed user manual (chapter [4]) is released together with the software, to guide the user in building and learning to use the library, and also to help him with as many as possible problems he could encounter using it. In case the user wants to put his hands on the source code, a full Doxygen documentation is available; the documentation, combined with the programming style kept with code readability in mind, should minimize the user effort to get aligned with the code development.

Finally in [5] we have tested the released software on previously studied (1.5) sample problems: in this way we have verified the correctness of the given implementation,
Conclusions

and we have validated its performance; moreover we have shown and numerically validated the $hp$-refinement algorithm itself, studied so far only from a theoretical point of view. For the monodimensional case, which has been deeply investigated, the test results show a great accordance with the theoretical predictions, confirming the goodness of our work. The multidimensional case, which, because of the different geometries and domain discretizations available in multi-D, is more complex and with a larger number of cases to study, has not been exhaustively eviscerated yet; anyway the first results obtained on 2D meshes are really promising, showing accordance with the predictions as for the monodimensional case.

A deeper investigation of the 2D case is left to future development, and it will need an improvement of the software in its input/output section; probably optimizations of the code will be needed too, especially in case a 3D extension of the software will be in plan; possible improvements are already expected and marked inside the source code (e.g., the optimization of the basis functions evaluation). Simultaneously the multidimensional case will require an additional theoretical analysis, in order to build new test cases.

Moreover, in future the library can be extended integrating it with libraries different from the ones considered so far, to optimize its performances and/or to expand its range of use.

Finally, the algorithm could be extended to non-binary trees, making it applicable to basically every possible mesh. A deep theoretical optimality analysis should be provided in this case; once done this, the implementation step would require very little effort, thanks to the generality principles adopted in the design of the program.
Bibliography


