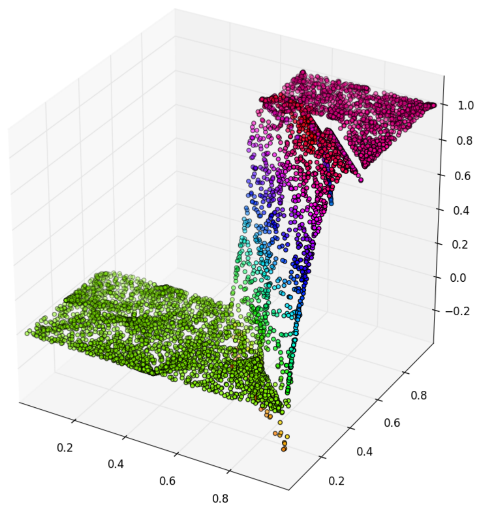
On *h*- and *hp*-type near-optimal tree generation and piecewise polynomial approximation in 1 and 2 dimensions

Jan Westerdiep
July 15, 2014

Thesis for BSc. Mathematics and BSc. Computer Science
Supervision: Prof.dr. R.P. Stevenson and dr. G.D. van Albada



Korteweg-de Vries Institute for Mathematics (KdvI)
Informatics Institute (IvI)
Faculty of Natural Sciences, Mathematics and Computer Science (FNWI)
Universiteit van Amsterdam

Abstract

In this Thesis, we will approximate $f \in L^2(D)$ for D one- or two-dimensional (a closed bounded interval in one dimension; a simple polygon in two) using piecewise polynomial approximations subject to some partition formed with dyadic refinements.

We will explore *near-optimal* tree generation: trees that yield piecewise polynomial approximations with the property that they are 'almost' as good as the best-possible piecewise polynomial approximation with the (up to a constant factor) same degrees of freedom.

At first, we will take the polynomial degree to be constant on each element of the resulting partition (h-type approximation) and generalize by making this a variable (hp-type approximation).

We will consider the one-dimensional case and formulate h- and hp-type near optimality and create approximations using the methods provided. It will become apparent that the two-dimensional formulation is more involving and we will introduce polygonal triangulation, orthogonal polynomials over a triangle and refinement of a triangle using Newest Vertex Bisection.

We will also cover an implementation in C of the theory and provide numerical results.

Title: On h- and hp-type near-optimal tree generation and piecewise polynomial approximation in 1 and 2 dimensions
Cover image: this is an approximation of a function using piecewise polynomials, created by the algorithms presented in this Thesis. The approximant is piecewise defined on a partition of triangles. The function approximated is

$$g : (0,1)^2 \setminus ((0,0.5) \times (0.5,1)) \to \mathbb{R} : (x,y) \mapsto \mathbb{1}_{\{x+y>1.1\}}.$$

Author: Jan Westerdiep, `jan.westerdiep@student.uva.nl`, 10219242.
Supervision: Prof.dr R.P. Stevenson, `r.p.stevenson@uva.nl`; dr. G.D. van Albada, `g.d.vanAlbada@uva.nl`.
End date: July 15, 2014.

Contents

Introduction

In this Thesis, we will shed light on a recent and interesting development in the area of *adaptive approximation*. Colloquially, adaptive approximation is the art of approximating a function by *adapting* the method to the problem at hand. This is applied in many fields, from numerical methods for solving differential equations to image processing. Usually, adaptive approximation methods can be described by a *tree* which records the adaptive decisions. Of course, many such trees exist and it is often important to find the *optimal* tree, i.e., the tree with minimal approximation error across some class of trees (for instance, the class of trees with less than n inner nodes). We could find this tree by considering all trees within this class. However, the number of such trees grows exponentially in n and we are therefore interested in quickly[1] finding *near-optimal* trees, a subject formally introduced later in this Thesis.

Our case

In our case, we have a function f for which we want to find piecewise polynomial approximations subject to a partition, i.e., restricted to any partition element, the approximant is a polynomial on this element. In order to efficiently encode this partition, we only consider certain refinements: elements of a partition are subject to some *subdivision rule* (e.g., if the element happens to be an interval $[a, b]$, the subdivision rule used in this Thesis refines this element to two intervals of equal length, namely $[a, (a + b)/2]$ and $[(a + b)/2, b]$). The question of *how* to refine an element is now reduced to *whether or not* to refine it.

Consider some f on $[0, 1]$. Subdividing $[0, 1]$ into $[0, 1/2], [1/2, 1]$ and subsequently $[0, 1/2]$ into $[0, 1/4], [1/4, 1/2]$ yields the partition

$$\{[0, 1/4], [1/4, 1/2], [1/2, 1]\}.$$

This partition can be also be written in tree form, see Figure 1. Refinement of a partition element is the same as adding children to a leaf node of this tree. In other words, partitions created like this can be identified with the leaves of some *partition tree.*

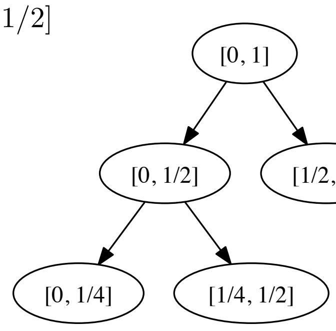


Figure 1: Example of a partition subject to some subdivision rule, written in tree form.

Assuming for the moment that finding the *polynomial of best approximation* over a partition element is "easy", the task left at hand is finding a good partition, or equivalently, how to grow the partition tree. In this Thesis, we will consider two types of trees and a respective refinement strategy for each.

[1]In complexity analysis, an algorithm is considered quick when its complexity is of polynomial order.

h- and hp-refinement

The h-refining algorithm, invented in 2004 [4] and improved in 2007 [8], is a method that, given tree T_0, adaptively grows T_{j-1} to T_j by choosing which leaves to subdivide. Furthermore, this tree is found using computations linear in n – the number of inner nodes of T_j. The degrees of freedom[2] of each node in the tree is constant. It was proven that the resulting tree T_j is near-optimal: the h-refining algorithm finds trees that yield approximations not worse (up to constant factor) than the best tree with a constant factor less inner nodes. Trees formed by this algorithm are referred to as h-trees, as the variable in these trees is h – the length of the subintervals.

It is known ([20]) that while functions that are discontinuous or otherwise 'non-smooth' are best approximated by such h-refining strategies, smooth functions benefit more from high-order polynomial approximations. It is therefore that the hp-refining algorithm was invented around 2013 [6] (and subsequently improved in 2014 [7]; it is as of yet still in active development). It is a generalization of h-refinement, allowing the degrees of freedom – referred to as p in literature – to vary between elements of the tree. In the h-case, the total degrees of freedom was controlled by choosing *which* elements to subdivide. In this case, we control them by allowing N degrees of freedom at iteration N of the algorithm. Thus, the hp-algorithm finds hp-trees T_N with total degrees of freedom equal to N. The method proposed is between $\mathcal{O}(N \log N)$ and $\mathcal{O}(N^2)$ in complexity depending on the function f, and is again near-optimal (in the hp sense: T_N is compared against the optimal hp-tree instead of h-tree).

Applications

One may wonder what the use of all this is. The main application of both methods described lies in the area of the Finite Element Method, one of the most widely used methods for finding numerical approximations of solutions to differential equations. While this is an area of research far beyond the scope of this thesis, we will sketch the main idea. FEM is the collective noun of all methods for connecting many simple equations over many small subdomains – named finite elements – to approximate a more complex equation over a larger domain. The solution to the equation is – contrasting to what is assumed in this thesis – unknown and information can be retrieved by means of numerical queries. Finite Element approximants are adaptively improved by means not unlike the above (e.g., marking elements for subdivision). On each finite element, an *a posteriori error bound* is measured, indicating "goodness of fit" on this particular element. Often times, these approximants contain a huge number of degrees of freedom, which hinders usability.

This is where the methods described above come into play. The Finite Element approximant is used as input and we will opt to find an approximation to this approximant with much fewer degrees of freedom, while retaining much of the goodness of fit. This yields a figure that looks like Figure 2. The solid lines represent error bounds found with Finite Element approximation. The dashed lines represents a step of the hp-refining algorithm applied on the approximant. We can see the convergence rate *without* our hp-algorithm in red. The green line represents the improved convergence rate found *with* use of the theory of this Thesis.

[2]Degrees of freedom is the number of parameters that may vary independently. In our case, we have a polynomial approximation on each element of the partition. In one dimension, the degrees of freedom of a polynomial of degree r is $r+1$, as $p(x) = a_0 + a_1 x + \dots a_r x^r$ has $r+1$ coefficients.

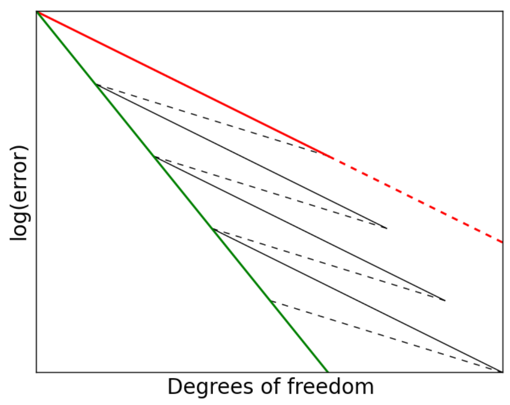


Figure 2: Idea of the application of the theory in this Thesis. The solid lines display Finite Element error bound progression. Dashed lines display invocation of the *hp*-algorithm to decrease the number of degrees of freedom while retaining goodness of fit. The red line displays convergence rate when using FEM approximation only. The green line displays the improved performance using a combination of both FEM and the theory of this Thesis.

Contents of this Thesis

In Chapter 0, we will cover all theory discussed in earlier classes such as Linear Analysis and Numerical Analysis (we will give a formal introduction to trees and subdivision rule, and look at polynomial approximation in one dimension). In Chapter 1, the above *h*- and *hp*-type algorithms will be discussed in depth.

The remainder of the theory part of this Thesis is devoted to a change in dimensionality: whereas in Chapter 0, f is assumed to be univariate and its domain to be an interval, in Chapter 2 we will discuss what happens when f becomes bivariate, with polygonal domain. We will glance over polygonal triangulation; a subdivision rule for triangles called Newest Vertex Bisection; discuss conforming partitions, and we will study the PKD-polynomials – a two-dimensional analogue to Legendre polynomials.

As this Thesis was written to conclude both a Bachelor's degree in Mathematics and in Computer Science, we will spend time not only theorizing the above algorithms. A great deal of time was devoted to actually implementing the theory of Part I (comprising Chapters 0 through 2). We will then experiment with the theory and implementation throughout Chapter 3, after which Chapter 4 will take on the details of the two-dimensional implementation, and we will use Mathematical and Computer Scientifical ingenuities to optimize this. Both source code optimizations and parallelization – the art of carrying out multiple calculations *simultaneously* – will receive attention. We will end the formal part with a discussion on possible improvements and points of future study.

We will start the Appendices with some non-essential theory regarding *conforming partitions*. The less-than-appetizing details of various other matters are left for Appendices B, C and D.

This Thesis is concluded with a Popular Synopsis or *Populaire Samenvatting* – a summary written in Dutch meant to be readable for people with a high-school educational level of Mathematics.

We wish the reader best of luck in the – admittedly sometimes terse or hard to read – journey ahead. *Bon voyage!*

Part I

Theory

0 Preliminaries

The following Chapter will cover relevant results found in earlier Bachelor courses such as Graph Theory, Linear Analysis and Numerical Analysis. This is also the reason that many of the sources provided in this Chapter are from the literature of these courses.

We will cover some basic results on Hilbert spaces in order to ease proofs and formulations regarding *polynomial approximation.* After this, we will glance over *pairing functions*, which will prove useful in the two-dimensional polynomial approximation of Chapter 2. We will then formally introduce the notion of a *tree* and *subdivision rule.*

We will conclude this preliminary Chapter by looking at the vast importance of polynomials in one dimension, and their role in approximation of functions.

0.1 Functional Analysis

The following results were found in [19]. This section is for the mathematically inclined reader and can be easily skipped.

Theorem 0.1 ([19, Thm. 1.61]). *If D is any domain in $\mathbb{R}^k$, $k \geq 0$, then $L^2(D)$ is a Hilbert space, with inner product*

$$\langle f, g\rangle := \int_D f(\mathbf{x})g(\mathbf{x})d\mathbf{x},$$

where the integral is k-dimensional.

Definition 0.2. For a linear subspace Y of an inner product space X, the *orthogonal complement of* Y is defined as

$$Y^{\perp} := \{x \in X : \langle x, y\rangle = 0 \quad \forall y \in Y\}.$$

Lemma 0.3 ([19, Lem. 3.30]). *Let Y be a linear subspace of an inner product space X. Then*

$$x \in Y^{\perp} \iff \|x\| = \inf_{y\in Y} \|x - y\|.$$

Theorem 0.4 ([19, Thm. 3.32]). *If A is a non-empty, closed and convex subset of a Hilbert space $\mathcal{H}$, then for every $p \in \mathcal{H}$ there is a unique $q \in A$ such that*

$$\|p - q\| = \inf_{a\in A} \|p - a\|.$$

This element q is called the best approximation of p with respect to A.

Theorem 0.5. *Let Y be a closed linear subspace of a Hilbert space $\mathcal{H}$. Then for $h \in \mathcal{H}$, $y \in Y$ is the best approximation of h with respect to Y iff $(h - y) \in Y^{\perp}$.*

Proof. The proof is a simple one-liner:

$$\|y - h\| = \inf_{y'\in Y} \|y' - h\| = \inf_{y'\in Y} \|y' - y + y - h\| \iff (y - h) \in Y^{\perp}. \qquad \square$$

Theorem 0.6 ([19, Thm. 3.22]). *If X is an inner product space of dimension k, with $\{e_1, \ldots, e_k\}$ an orthonormal basis, then*

$$\left\|\sum_{n=1}^{k} \alpha_n e_n\right\|^2 = \sum_{n=1}^{k} |\alpha_n|^2, \quad \forall\{\alpha_1, \ldots \alpha_k\} \in \mathbb{R}^k.$$

Theorem 0.7 ([19, Thm. 3.34]). *Let Y be a closed linear subspace of a Hilbert space $\mathcal{H}$. For $x \in \mathcal{H}$, there are unique $y \in Y, z \in Y^{\perp}$ such that $x = y + z$ and $\|x\|^2 = \|y\|^2 + \|z\|^2$.*

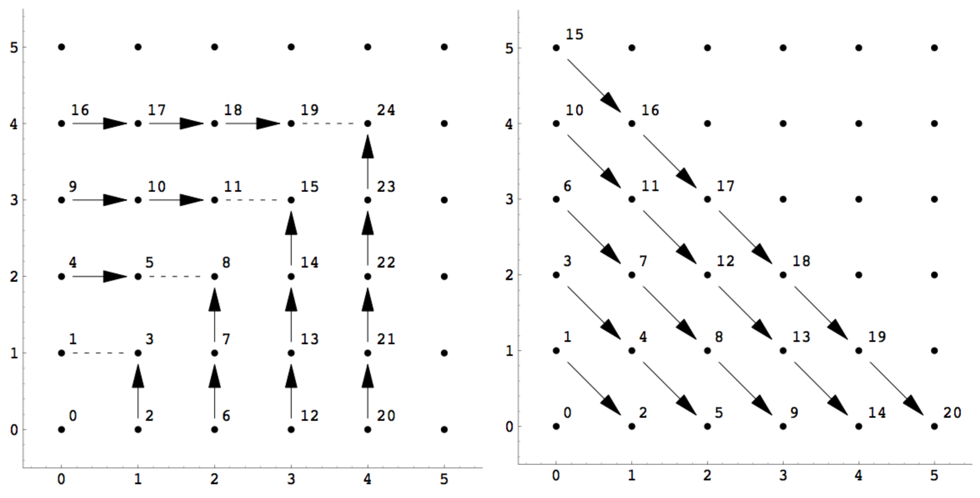


Figure 1: Left: the Szudzik pairing function. Right: the Cantor pairing function.

0.2 Pairing functions

It is widely known that there exist many bijections between $\mathbb{N} \times \mathbb{N}$ and $\mathbb{N}$. Such bijective maps are called *pairing functions.* We present two pairing functions, both with different applications.

0.2.1 Szudzik pairing function

Definition 0.8. The Szudzik pairing function was formulated in 2006[25] and is defined by

$$\pi(x, y) := \begin{cases} y^2 + x & x < y \\ x^2 + x + y & x \geq y \end{cases}.$$

It traverses the squares inside-out (see Figure 1). The inverse is formed by

$$\pi^{-1}(z) = \begin{cases} (z - \lfloor\sqrt{z}\rfloor^2, \lfloor\sqrt{z}\rfloor) & z - \lfloor\sqrt{z}\rfloor^2 < \lfloor\sqrt{z}\rfloor \\ (\lfloor\sqrt{z}\rfloor, z - \lfloor\sqrt{z}\rfloor^2 - \lfloor\sqrt{z}\rfloor) & z - \lfloor\sqrt{z}\rfloor^2 \geq \lfloor\sqrt{z}\rfloor \end{cases}.$$

In this thesis, the Szudzik pairing function will be used to store a square matrix in the form of a one-dimensional array.

0.2.2 Cantor pairing function

Definition 0.9. The Cantor pairing function is defined by

$$\pi(x, y) := (x + y)(x + y + 1)/2 + y$$

and traverses the triangle above over the diagonals (see Figure 1). The inverse is formed by

$$\pi^{-1}(z) = (x, y), \quad w = \left\lfloor \frac{\sqrt{8z+1} - 1}{2} \right\rfloor, \quad t = \frac{w^2 + w}{2}, \quad y = z - t, \quad x = w - y.$$

In this thesis, the Cantor pairing function is used to provide a counting for a set of the triangular form in Figure 1.

0.3 Trees

Definition 0.10. A *tree* is a connected directed graph $G = (T, E)$ without any cycles.

A vertex in a tree is often called a *node*. The node with only outgoing edges is called the *root node*, D. Trees with root nodes are called *rooted trees*.

Definition 0.11. In a rooted tree, each node has a *depth* d – the length of the path to the root. Given a node $\triangle \neq D$, one can define the *parent* to be the node $\triangle^+ \in T$ with $d(\triangle) = d(\triangle^+) + 1$ such that $(\triangle^+, \triangle) \in E$. On the other hand, the *children* of $\triangle$ – `children`$(\triangle)$ – are all the nodes that have $\triangle$ as their parent. Nodes that share a parent are called *siblings*. Nodes without children are called *leaves* (with `leaves`(T) denoting the set of leaves of T) and nodes that are not leaves are called *inner nodes* (with `inners`(T) the set of inner nodes).

Lemma 0.12. *In a rooted tree, every node except for D has a parent.*

With the notion of trees properly introduced, we can now further enlarge our toolset.

Definition 0.13. A *subdivision rule* `subdivide` is a way to partition a given set $\triangle \subset D$ into 2 elements.

Example 0.14. A possible subdivision rule on $[a, b] \subset \mathbb{R}$ is to divide this interval into two equal halves: $[a, b]$ becomes $\{[a, \frac{a+b}{2}], [\frac{a+b}{2}, b]\}$.

Notation 0.15. Strictly speaking, a tree is an ordered pair (T, E). Because the edges are in our case easy to determine (by the implied parent-child relation), we will loosen our terminology and say that a tree *equals* its set of vertices T.

If we take some interval $D := [a, b] \subset \mathbb{R}$ and identify this with a tree containing only the root node D, we can create trees by means of the above subdivision rule. Subdivision of D yields two elements that together form a partition of D. This can be seen as creating two children of the root node. Of course, we can do this again with one of the resulting children.

This way, we can create a plethora of binary trees of which all leaves are subsets of D. The attentive reader might have already discovered that these leaves hold a the following property, which holds even for domains of higher dimensions.

Lemma 0.16. *Let $D \subset \mathbb{R}^k$. For a tree T generated by iterated use of the subdivision rule, it holds that the set of leaves of T is a partition of D.*

Remark. It is useful to point out that trees T of the above type can create *adaptive* partitions of D. This means that in some regions of D, a higher concentration of elements might be present. This means that we can *adapt* our tree generation algorithm to the problem at hand, creating a partition well suited for this particular problem.

0.4 Polynomial approximation in one dimension

Definition 0.17. A *polynomial* is a function $p : \mathbb{R} \to \mathbb{R}$ of the following form:

$$p(x) = \sum_{k=0}^{n} a_k x^k.$$

The values a_k are the *coefficients* of p, and the largest power of x with nonzero coefficient is called the *degree* of p. As a polynomial of degree n has $n + 1$ coefficients, the number of *degrees of freedom* is $n + 1$. The set of all polynomials is denoted by $\mathcal{P}$. When constrained to a domain $D \subset \mathbb{R}$, $\mathcal{P}(D) := \{q_{|D} : q \in \mathcal{P}\}$ denotes the set of all polynomials from D to $\mathbb{R}$.

These polynomials are useful instruments in dealing with function approximation. Given an interval $D = [a, b] \subset \mathbb{R}$ and a Lebesgue space $L^p(D)$, with $1 \leq p \leq \infty$, the following results hold and show the vast importance of polynomials.

Theorem 0.18. *For $1 \leq p < \infty$, the set $\mathcal{P}(D)$ is dense in $L^p(D)$.*

Proof. Our interval D is closed and bounded, thus by [19, Thm. 1.35] compact. Then by [19, Thm. 1.40], the set $\mathcal{P}(D)$ is dense in $C(D)$, the set of continuous functions from D to $\mathbb{R}$. Lastly, by [19, Thm. 1.62], $C(D)$ is dense in $L^p(D)$. Transitivity of denseness implies the result. $\square$

Theorem 0.19 ([24, Thm. 8.1])**.** *The set $\mathcal{P}(D)$ is dense in $C(D)$ with respect to the ∞-norm $\|\cdot\|_\infty$.*

Remark 0.20. In the above theorem, we deliberately did not state that $\mathcal{P}(D)$ lies dense in $L^\infty(D)$! This would imply that $\overline{C(D)} = L^\infty(D)$. To see this cannot be the case, consider $D = [-1, 1], f(x) = \operatorname{sgn}(x)$. Any continuous function g with $\|f - g\|_\infty < \frac{1}{3}$ must have $g(x) < f(x) + \frac{1}{3} = -\frac{2}{3}$ for $x < 0$. Analogously, $g(x) > \frac{2}{3}$ for $x > 0$. By continuity, $g(0)$ must satisfy both $g(0) \leq -\frac{2}{3}$ and $g(0) \geq \frac{2}{3}$. Such a function cannot exist, and so we cannot get arbitrarily "close" to f using only continuous functions.

We will look at two special cases, namely $p = \infty$ and $p = 2$. We will see that $p = \infty$, while perhaps the most intuitive, has little practical use. Setting $p = 2$ yields the space of square-integrable functions $L^2(D)$. From Theorem 0.1 we know that this space holds a very special property in the sense that it is a Hilbert space. This means we can use the theory provided in §0.1.

0.4.1 ∞-norm

Given an $\epsilon > 0$ and a continuous f, we found with Theorem 0.19 that it is possible to find a polynomial p such that

$$\|f - p\|_\infty := \sup\{|(f - p)(x)| : x \in D\} < \epsilon.$$

If we however restrict ourselves to just polynomials of degree n or less (denoted by $\mathcal{P}^n$), this result no longer holds. It is therefore interesting to look at the lowest value that *is* obtainable, or more precisely:

$$\text{Given } f \in C(D) \text{ and } n \geq 0, \text{ find } p_n \in \mathcal{P}^n \text{ such that}$$
$$\|f - p_n\|_\infty = \inf_{q \in \mathcal{P}^n} \|f - q\|_\infty. \tag{1}$$

Theorem 0.21 ([24, Thm. 8.2])**.** *The equality in* (1) *is attained for a polynomial of degree n, i.e., the infimum is a minimum.*

Because this p_n *minimizes* the *maximal* absolute value of $f(x) - q(x)$, this polynomial is often referred to as the *minimax polynomial.*

Theorem 0.22 ([24, Thm. 8.5])**.** *For $f \in C(D)$, there exists a unique minimax polynomial p_n.*

While we haven't shown the proof of Theorem 0.21 here, the reader is invited to read it and conclude that this proof is not *constructive* in the sense that it gives no *algorithm* to find the minimax polynomial. This makes for lesser practical use, as an implementation would like to actually *find* this polynomial. The most widely used solution is an *iterative* algorithm[1] – the Remez Algorithm [1] – that can find minimax approximations under certain conditions of f. However, we will not look into this in more detail.

[1]An iterative algorithm creates a sequence of improving approximate solutions to some problem.

0.5 2-norm

As stated earlier, the set of polynomials $\mathcal{P}$ is dense in $L^2(D)$. When we confine ourselves to using polynomials of degree n or less, we get a problem analogous to (1):

$$\text{Given } f \in L^2(D) \text{ and } n \geq 0, \text{ find } p_n \in \mathcal{P}^n \text{ such that}$$
$$\sqrt{\int_D [f(x) - p_n(x)]^2 dx} =: \|f - p_n\|_2 = \inf_{q \in \mathcal{P}^n} \|f - q\|_2. \tag{2}$$

Theorem 0.23. *Given $f \in L^2(D)$, there exists a unique polynomial $p_n \in \mathcal{P}^n(D)$ that solves Problem (2). This polynomial is called the* polynomial of best approximation*.*

Proof. As $L^2(D)$ is a Hilbert space, Theorem 0.4 applies. □

Theorem 0.24. *For $f \in L^2(D), p_n$ is the polynomial of best approximation iff*

$$\langle f - p_n, q \rangle = 0, \quad \forall q \in \mathcal{P}^n, \tag{3}$$

or in other words, $f - p_n$ is orthogonal to every element in $\mathcal{P}^n$.

Proof. Use Theorem 0.5. □

Example 0.25. We can easily find this *polynomial of best approximation* $p_n(x) = c_0 + \cdots + c_n x^n$, as will become apparent in this argument.

For simplicity, assume $D = [0, 1]$. Using Theorem 0.24 and noting that $x^j \in \mathcal{P}^n$ for $0 \leq j \leq n$, we find that

$$0 = \langle f - p_n, x^j \rangle = \int_0^1 [f(x) - p_n(x)] x^j dx$$

so

$$\int_0^1 f(x) x^j dx = \sum_{k=0}^n c_k \int_0^1 x^{j+k} dx = \sum_{k=0}^n \frac{c_k}{j+k+1}.$$

for all j. This leads to a system of $n + 1$ linear equations:

$$\sum_{k=0}^n H_{jk} c_k = b_j, \quad j = 0, \ldots, n, \quad H_{jk} = \int_0^1 x^{k+j} dx = \frac{1}{k+j+1}, \quad b_j = \int_0^1 f(x) x^j dx. \tag{4}$$

The matrix H_{jk} has nonzero determinant [10], so this system has a unique solution $(c_0, \ldots, c_n)$. This matrix is often referred to as the Hilbert matrix of dimension $n + 1$. There is however a problem with this construction: as we are solving a linear system, we are effectively computing the inverse of the Hilbert matrix.

This inverse has integer elements [10] and for $n \geq 14$, cannot be represented by 64-bit integers [29]. The problem lies in the choice of basis, as the basis $\{x^j : 0 \leq j \leq n\}$ is highly ill-conditioned. To formalize this, consider the following.

0.5.1 Bases and conditioning

Definition 0.26. A *basis* over a finite-dimensional space V is a set of linearly independent elements $\{v_j : 1 \leq j \leq \dim(V)\}$ that spans V.

Definition 0.27. The *mass matrix* of a basis $\{\varphi_j : j \leq n\}$ over $\mathcal{P}^n$ on D is defined as

$$M = \left[\int_D \varphi_i(x)\varphi_j(x)dx\right]_{i,j\leq n}.$$

Remark 0.28. As $\varphi_i(x)\varphi_j(x) = \varphi_j(x)\varphi_i(x)$, M is a symmetric matrix.

Definition 0.29. The *condition number* of a matrix A is defined as

$$\kappa_2(A) = \|A\|\|A^{-1}\| \geq 1, \quad \text{where } \|\cdot\| = \|\cdot\|_2,$$

and measures the factor with which the solution $x = A^{-1}b$ of the linear system $Ax = b$ can change for a small change in A or b (e.g., rounding errors due to finite precision). A condition number of 1 means that errors do not inflate; this is the best-obtainable value.

Theorem 0.30 ([24, p. 73]). *The condition number of the $n \times n$ mass matrix* (4) *of the basis $\{x^j : j \leq n-1\}$ of monomials grows like $\frac{4^n}{\sqrt{n}}$.*

This condition number is enourmous, and as we will see, unnecessarily large.

0.5.2 Orthogonal bases

Definition 0.31. A basis $\{\varphi_j : j \leq n\}$ over $\mathcal{P}^n$ is called an *orthogonal basis* on D if the following condition holds:

$$\int_D \varphi_j(x)\varphi_k(x)dx = 0 \text{ if } k \neq j.$$

We will now show that we can find an orthogonal basis over $\mathcal{P}^n$ on any domain $D = (a, b)$, by means of the so-called *Gram-Schmidt orthogonalization.*

Let $\varphi_0(x) = 1$ and suppose φ_j has already been constructed for j up to $n \geq 0$. Then

$$\int_a^b \varphi_k(x)\varphi_j(x)dx = 0, \quad k \in \{0, \ldots, j-1, j+1, \ldots, n\}.$$

Define

$$q_{n+1}(x) = x^{n+1} - a_0\varphi_0(x) - \cdots - a_n\varphi_n(x), \quad a_j = \frac{\int_a^b x^{n+1}\varphi_j(x)dx}{\int_a^b \varphi_j^2(x)dx}.$$

It follows that

$$\begin{aligned}\int_a^b q(x)\varphi_j(x)dx &= \int_a^b x^{n+1}\varphi_j(x)dx - a_j\int_a^b \varphi_j^2(x)dx \\ &= 0, \quad 0 \leq j \leq n\end{aligned}$$

by using orthogonality. With this choice of a_j, we have ensured that q_{n+1} is orthogonal to all previous members of the basis, and φ_{n+1} can be defined as any nonzero multiple of q_{n+1}.

We conclude with a direct way to solve problem (2).

Theorem 0.32. *Given degree n, interval $\triangle = [a, b]$ and orthogonal basis $\{\varphi_j : j \leq n\}$ over $\mathcal{P}^n(\triangle)$, and a function $f \in L^2(\triangle)$, the polynomial*

$$p_n(x) = \gamma_0\varphi_0(x) + \cdots + \gamma_n\varphi_n(x), \quad \gamma_j = \frac{\int_a^b f(x)\varphi_j(x)dx}{\int_a^b \varphi_j^2(x)dx}$$

is the unique polynomial of best approximation of degree n to f.

Proof. We use Theorem 0.24. If the result holds for every basis function φ_j, then it must hold for every q. We see that

$$\forall j \leq n, \quad \langle f - p_n, \varphi_j \rangle = \langle f, \varphi_j \rangle - \sum_{k=0}^{n} \gamma_k \langle \varphi_k, \varphi_j \rangle = \langle f, \varphi_j \rangle - \gamma_j \langle \varphi_j, \varphi_j \rangle = 0. \qquad \square$$

0.5.3 Legendre polynomials

A direct application of the above theory is the basis of the *Legendre polynomials*, the basis of choice in one-dimensional approximation. We wish to create an orthogonal basis for the interval $(-1, 1)$ for all n.

Beginning with $P_0(x) = 1$, we find $P_1(x) = q_1(x) = x$. Continuing the Gram-Schmidt procedure, we find

$$P_0(x) = 1, \quad P_1(x) = x, \quad P_2(x) = \frac{1}{2}(3x^2 - 1), \quad P_3(x) = \frac{1}{2}(5x^3 - 3x), \quad \ldots.$$

Replacing x by

$$\frac{2}{b-a}x + \frac{a+b}{a-b} \tag{5}$$

yields an orthogonal basis $\{\tilde{P}_n\}$ over (a, b) which we will call the *shifted Legendre polynomials.*

Theorem 0.33 ([3, p. 743])**.** *The Legendre polynomials P_n satisfy the following recurrence relation:*

$$(n+1)P_{n+1}(x) = (2n+1)xP_n(x) - nP_{n-1}(x)$$

Theorem 0.34 ([19, Ex. 3.28])**.** *The n'th Legendre polynomial satisfies the following equality:*

$$\int_{-1}^{1} P_n(x)^2 dx = \frac{2}{2n+1}.$$

More generally, the n'th shifted Legendre polynomial satisfies

$$\int_{a}^{b} \tilde{P}_n(x)^2 dx = \frac{b-a}{2n+1}.$$

We will now argue that the condition number of the Legendre polynomial basis is much better than that of the monomial basis.

Theorem 0.35. *If A is a real symmetric matrix, then $\kappa_2(A) = \frac{\max(|\lambda_i|)}{\min(|\lambda_i|)}$, where λ_i is the i'th Eigenvalue of A.*

Theorem 0.36. *After normalization, the condition number of the $n \times n$ mass matrix of the Legendre basis of Theorem 0.33 is 1.*

Proof. As the Legendre polynomials are orthogonal, the mass matrix M is diagonal. After normalization, $M = I_n$. Then, by Theorem 0.35, we find $\kappa_2(M) = 1$. $\square$

1 One-dimensional tree generation

In §0.5, we saw that given high degree r, we can approximate any function arbitrarily well (in L^2-sense). Given some domain $D \subset \mathbb{R}$ and function in $L^2(D)$, we can approximate f over polynomials of degree $0, 1, 2, \dots$, until some tolerance is reached. This is called p-refinement, as the degree of the polynomial approximant – p – is refined[1].

In Lemma 0.16, we concluded that every tree T formed by the subdivision rule induces a partition of D – namely the set of its leaves. Now, let r be fixed, and for each leaf, approximate f with degree r using its polynomial of best approximation. The result is a piecewise polynomial approximation of f (subject to the tree T), with fixed polynomial degree over each leaf. If we now decide to subdivide some leaf, we get a finer partition and thus a better approximation. Iterated use of this scheme yields a type of approximation called h-type approximation, as the length of the intervals – h – is adaptively refined in each step.

To abstract away from one-dimensional polynomial approximation and present this chapter in a more general setting, we reformulate as follows. Imagine some domain D, not necessarily a subset of $\mathbb{R}$. Take some list of nested function spaces $V_1 \subset V_2 \subset \cdots$, where V_r is an r-dimensional subspace of $L^2(D)$ such that $\lim_{r\to\infty} V_r = L^2(D)$.[2] We say that each element in V_r has r *degrees of freedom*. Now imagine some function $f \in L^2(D)$. The theory in §0.1 tells us that for every r, there is a unique element of best approximation in V_r to f.

We create a tree with only a root node – D – and approximate f with the above unique element. We then subdivide this root node to receive a tree with two leaves, and repeat the scheme of the paragraph above. The result is again a tree with h-refinements, also known as an h-tree.

Of course, many possibilities exist to create such an h-tree and often, one desires to find the *optimal* tree with N subdivisions. One way to find this optimal tree is to traverse all possible trees, but seeing there are around 2^N such trees, this quickly becomes impossible. We are therefore interested in finding a *near-optimal* tree (a concept formally introduced shortly) by means of a quick algorithm (i.e., of polynomial complexity). We will see that there exists a non-trivial algorithm that finds near-optimal h-trees in linear time.

1.1 h-tree generation

Definition 1.1. The function $e : T \to \mathbb{R}_{\geq 0}$ is a function assigning the best obtainable error by approximating f on $\triangle \in T$ with functions in V_r. If this function further satisfies the *subadditivity rule*

$$e(\triangle) \geq e(\triangle') + e(\triangle''), \quad \{\triangle', \triangle''\} = \texttt{children}(\triangle),$$

then it is called an *error functional.*

Example 1.2. We will choose e to be

$$e(\triangle) := \|f - p_\triangle\|^2_{2,\triangle} = \min_{q\in\mathcal{P}^{r-1}} \|f - q\|^2_{2,\triangle},$$

[1]Literature is sometimes inconsistent with their nomenclature. Both p and r are often used to denote polynomial degree. In the following, we will use r.

[2]This is a lot less freightening than it may look at first sight. Take $D \subset \mathbb{R}$ and V_r to be the set of polynomials of degree $r - 1$ over D. Then the result holds.

the square of the difference in 2-norm of f with the best polynomial approximation over $\triangle$. This square is needed to ensure satisfaction of the subadditivity rule, and $r-1$ is used to create an r-dimensional space.

Using the theory from the preliminary Chapter 0, we can find $e(\triangle)$ using the shifted Legendre basis applied to Theorem 0.32, together with the recurrence relation from Theorem 0.33. This gives us $p_\triangle$, and integration yields $e(\triangle)$.

Definition 1.3. Given an error functional e, we can define the *total error* $E(T)$ of a tree as

$$E(T) := \sum_{\triangle \in \texttt{leaves}(T)} e(\triangle).$$

Remark 1.4. Note that because of the subadditivity rule, it must hold that

$$T_1 \supset T_2 \implies E(T_1) \leq E(T_2).$$

This means that we can never increase our total error by refining the partition.

Given the functionals e and E, and $m \in \mathbb{N}$, consider the class $\mathcal{T}_m$ of all trees T generated from D with $\#\texttt{inners}(T) \leq m$ inner nodes. Define

$$E_m := \min_{T \in \mathcal{T}_m} E(T)$$

as the best obtainable error by m subdivisions. Of course, this error can be explicitly found by considering all $\mathcal{O}(2^m)$ trees in $\mathcal{T}_m$. This is however an exponential problem and therefore not suitable for real-life application.

Definition 1.5. A h-tree generating algorithm is called *near-optimal* if it adaptively finds trees T_j such that there exist constants $0 < C_1 \leq 1 \leq C_2$ independent of f with

$$m \leq C_1 \#\texttt{inners}(T_j) \implies E(T_j) \leq C_2 E_m.$$

In the following, we will look at several algorithms. Each algorithm will operate in the same way: we have a tree T_j with $\#\texttt{inners}(T_j)$ inner nodes, and want to find which leaves to subdivide in order to generate T_{j+1}.

1.1.1 The naive approach

With our error functional e and leaves in place, why not just subdivide the leaves where the error is maximal? It might seem like a valid point at first.

Algorithm 1.6 (Greedy)**.** *For $j = 0$, $T_0 = D$ is the root node of $\mathcal{T}^*$. If T_j has been defined, examine all leaves $\triangle \in \texttt{leaves}(T_j)$ and subdivide the leaves $\triangle$ with largest $e(\triangle)$ to produce T_{j+1}. In case of multiple candidates, subdivide all.*

This algorithm is called *greedy*, for it makes the best *local* choice, rather than considering the *global* problem at hand. We refer to Appendix C to prove that Algorithm 1.6 is not near-optimal.

1.1.2 Binev 2004

The above algorithm shows a simple concept: adaptive refinement of a partition. The tree this approach generates is thusly called an h-tree for it refines h, the mesh grid size.

Instead of looking purely at the error functional e in Algorithm 1.6, a *modified error functional* is introduced in [4]. This modified error functional $\tilde{e}$ penalizes nodes that don't improve after a subdivision. This modification makes for provable performance enhancements, as we will shortly see.

Let $\tilde{e}(D) := e(D)$. Then, given that $\tilde{e}(\triangle)$ is defined, let $\tilde{e}$ for both children $\triangle_j$ of $\triangle$ be as follows:

$$\tilde{e}(\triangle_j) := \frac{e(\triangle_1) + e(\triangle_2)}{e(\triangle) + \tilde{e}(\triangle)} \tilde{e}(\triangle). \tag{1.1}$$

Algorithm 1.7 (Binev2004[4, p. 204])**.** *For $j = 0$, $T_0 = D$ is the root node of $\mathcal{T}^*$. If T_j has been defined, examine all leaves $\triangle \in$ `leaves`(T_j) and subdivide the leaves $\triangle$ with largest $\tilde{e}(\triangle)$ to produce T_{j+1}. In case of multiple candidates, subdivide all.*

Theorem 1.8 ([4, Thm. 5.2], [8, Thm. 2])**.** *Let $n := \#$`inners`(T_j) be the number of inner nodes of T_j. Algorithm 1.7 is near-optimal with parameters*

$$C_1 = 1/6, \quad C_2 = 1 + \frac{2(m+1)}{n+1-m}.$$

Furthermore, the algorithm uses up to $C_2(n+1)$ arithmetic operations and computations of e to find this T_j.

1.1.3 Binev 2007: A better modified error functional

The upper bound of Algorithm 1.7 can be improved by replacing the definition of the modified error in (1.1) in the following way. Let $\tilde{e}(D) := e(D)$ still, but for each child $\triangle_j \in$ `children`$(\triangle)$, let

$$\tilde{e}(\triangle_j) := \begin{cases} \left(\frac{1}{e(\triangle_j)} + \frac{1}{\tilde{e}(\triangle)}\right)^{-1} & \min\{e(\triangle_j), \tilde{e}(\triangle)\} > 0 \\ 0 & \text{else.} \end{cases} \tag{1.2}$$

The accompanying algorithm is now an exact restatement of the earlier result.

Algorithm 1.9 (Binev2007[8])**.** *For $j = 0$, $T_0 = D$ is the root node of $\mathcal{T}^*$. If T_j has been defined, examine all leaves $\triangle \in$ `leaves`(T_j) and subdivide the leaves $\triangle$ with largest $\tilde{e}(\triangle)$ to produce T_{j+1}. In case of multiple candidates, subdivide all.*

Theorem 1.10 ([8, Thm. 4])**.** *Using terminology of Theorem 1.8, Algorithm 1.9 is near-optimal with parameters*

$$C_1 = 1, \quad C_2 = 1 + \frac{m+1}{n+1-m}.$$

Remark 1.11. Comparing Theorem 1.8 with the above, we conclude that the new algorithm has larger C_1 and smaller C_2, and thus could be considered better.

1.1.4 Ensuring linear complexity

Algorithms 1.7 and 1.9 have linear complexity in the sense that the amount of operations needed to find T_j is linear in the number of subdivisions (or equivalently, the number of inner nodes). However, at each step, we are to subdivide the node $\triangle$ with highest modified error $\tilde{e}(\triangle)$. This means that we will want to keep a list of $(\triangle, \tilde{e}(\triangle))$ pairs, sorted by $\tilde{e}(\triangle)$ in order to make quick decisions. Keeping this list sorted across iterations means using some sorting algorithm, inherently a problem of complexity $\mathcal{O}(n \log n)$.

To overcome this problem, it is suggested to use binary bins: for each node, find an integer κ such that $2^\kappa \leq \tilde{e}(\triangle) < 2^{\kappa+1}$ and place $\triangle$ in bin κ. The highest nonempty bin now becomes the set of nodes to subdivide. Theorem 1.8 still holds true in this case ([4, p. 207]), with C_2 multiplied by 2 ([7, Rem. 2.2]).

Remark. While it may seem important to ensure linear complexity, in real-life applications the time spent computing $e(\triangle)$ severly overshadows the time spent sorting of a (relatively small) list, making this optimization less important.

1.2 *hp*-tree generation: Binev 2013

Adaptive approximation by piecewise polynomials can be generalized in different ways. One of the most investigated forms of it is the *hp*-approximation, in which the local size of elements of the partition *and* the degree of the polynomials may vary, but the total number of degrees of freedom is controlled.

One motivation for this generalization is that under certain conditions, both *h*- and *p*-refinement at best exhibit *algebraical* convergence rate, in the sense that the error is polynomial in the degrees of freedom[20] – $E_N \sim N^{-\alpha}$, while *hp*-adaptivity can achieve *exponential* convergence rate[20, Thm. 4.63] – $E_N \sim e^{-bN^\alpha}$. This is exactly why Binev introduced an *hp*-tree generating algorithm. We will generalize [7] a bit further (again, to abstract away from one-dimensional polynomial approximation) and assume not polynomials of degree r, but functions from some r-dimensional space V_r as described at the start of this Chapter.

Remark 1.12. As this algorithm is still in active development, a number of revisions of the algorithm have come available while writing this Thesis. One of the most recent changes involved a complete overhaul of the presentation, making it infinitely more understandable. For the previous algorithm, see [6].

We are now both refining the mesh and the degrees of freedom, so there is need for a degree-dependent error functional.

Definition 1.13. The function e_r is a function

$$e_r : \mathbb{N} \times T \to \mathbb{R}_{\geq 0}$$

assigning the best obtainable error by approximating f on $\triangle \in T$ with functions in V_r.

If this function further satisfies the subadditivity rules

$$e_r(\triangle) \geq e_{r+1}(\triangle) \qquad \text{and} \qquad e_1(\triangle) \geq e_1(\triangle') + e_1(\triangle''), \quad \{\triangle', \triangle''\} = \texttt{children}(\triangle),$$

then this function is called the *local error of approximation.*

Example 1.14. In the one-dimensional case, the local error of approximation used is the square of the L^2-norm of the difference between f and the approximating polynomial p_{r-1}:

$$e_r(\triangle) := \|f - p_{r-1}\|_{2,\triangle}^2 = \inf_{q \in \mathcal{P}^{r-1}} \|f - q\|_{2,\triangle}^2.$$

In essence, this is the same functional as $e(\triangle)$ in Example 1.2, viewing r as a function parameter instead of a constant. Therefore, computation is exactly analogous.

Given a tree T, we assign to every $\triangle \in \texttt{leaves}(T)$ a *node-specific* degree $r(\triangle)$. This gives rise to the set of nodal degrees

$$R(T) := (r(\triangle))_{\triangle \in \texttt{leaves}(T)},$$

and a *total error*

$$E(T, R(T)) := \sum_{\triangle \in \texttt{leaves}(T)} e_{r(\triangle)}(\triangle).$$

We define $\#R(T)$ to be the total number of degrees of freedom in T:

$$\#R(T) := \sum_{\triangle \in \texttt{leaves}(T)} r(\triangle).$$

Given $m \in \mathbb{N}$, the best-obtainable *hp*-adaptive error with m degrees of freedom is

$$E_m^{hp} := \min\{E(T, R(T)) : \#R(T) \leq m\}.$$

In light of the previous, one would desire an algorithm to be *near-optimal* in the sense that it adaptively finds pairs $(T_N, R(T_N))$ with degrees of freedom $\#R(T_N) = N$, such that there are absolute constants $C_1, C_2 > 0$ with

$$m \leq C_1 N \implies E(T_N) \leq C_2 E_m^{hp}.$$

1.2.1 Idea for an algorithm

To effectively describe an algorithm that fulfills this desire, one immensely profits from the following realisation. First, for a tree T, let $T_\triangle$ denote its largest subtree with root $\triangle$.

We will keep track of two trees: an *h*-tree $\mathcal{T}_N$ and an *hp*-tree T_N. The connection between the two is that $\mathcal{T}_N$ has exactly N leaves, whereas T_N is a subtree of $\mathcal{T}_N$ in such a way that, for every leaf $\triangle \in \texttt{leaves}(T_N)$,

$$r(\triangle) =: r(\triangle, \mathcal{T}_N) = \#\texttt{leaves}(\mathcal{T}_{N,\triangle}).$$

Let $\mathcal{T}_0$ equal D. At step N, we will subdivide a leaf of the current tree $\mathcal{T}_{N-1}$ to create an *h*-tree $\mathcal{T}_N$ with $\texttt{leaves}(\mathcal{T}_N) = N$ leaves. The method of finding this exact subtree will be shortly discussed.

The above has the consequence that we can identify the *hp*-tree with either $(T_N, \mathcal{T}_N)$ or $(T_N, R(T_N))$, as in both tuples the leaves have correctly defined degrees. A consequence is that the total error $E(T_N, \mathcal{T}_N)$ and $E(T_N, R(T_N))$ can be identified with each other as well.

The realisation leads to the following 'idea of an algorithm'. To create the *hp*-tree T_N, we find a *h*-tree $\mathcal{T}_N$ with $\#\texttt{leaves}(\mathcal{T}_N) = N$ leaves and define T_N to be that subtree that has minimal total error:

$$\mathcal{T}_N = \operatorname{argmin}\{E(T_N, \mathcal{T}_N) : \#\texttt{leaves}(\mathcal{T}_N) = N, T_N \subset \mathcal{T}_N\}.$$

The tree $\mathcal{T}_{N+1}$ is then formed from $\mathcal{T}_N$ in a way similar to Algorithm 1.9 using modified errors.

1.2.2 Finding T_N from $\mathcal{T}_N$

We will now create some useful functionals to find this subtree $T_N \subset \mathcal{T}_N$.

Definition 1.15. The *local hp-error* e^{hp} of a node $\triangle \in \mathcal{T}_N$ is recursively defined as

$$e^{hp}(\triangle, \mathcal{T}_N) := e_1(\triangle), \quad \triangle \in \texttt{leaves}(\mathcal{T}_N)$$

and for $\triangle \in \texttt{inners}(\mathcal{T}_N)$ with $\{\triangle', \triangle''\} := \texttt{children}(\triangle)$,

$$e^{hp}(\triangle, \mathcal{T}_N) := \min\{e^{hp}(\triangle', \mathcal{T}_N) + e^{hp}(\triangle'', \mathcal{T}_N), e_{r(\triangle, \mathcal{T}_N)}(\triangle)\}.$$

To find T_N, we save $\mathcal{T}_N$ as-is for future reference, and traverse $\mathcal{T}_N$ in a fine-to-coarse manner, trimming those nodes $\triangle$ for which $e^{hp}(\triangle, \mathcal{T}_N)$ equals $e_{r(\triangle,\mathcal{T}_N)}(\triangle)$, indicating that a higher polynomial order yields smaller total error than refining the grid.

It is important to note that the dependence of $e^{hp}(\triangle, \mathcal{T}_N)$ is on $\mathcal{T}_{N,\triangle}$ – the subtree of $\mathcal{T}_N$ rooted at $\triangle$ – only. This quantity only changes when $T_{N,\triangle}$ grows. It is therefore useful to define $e^{hp}_j(\triangle)$ to be $e^{hp}(\triangle, \mathcal{T}_N)$ whenever $j = r(\triangle, \mathcal{T}_N)$, or in other words,

$$e^{hp}_1(\triangle) := e_1(\triangle), \quad e^{hp}_j(\triangle) := \min\{e^{hp}_{r(\triangle',\mathcal{T}_N)}(\triangle') + e^{hp}_{r(\triangle'',\mathcal{T}_N)}(\triangle''), e_j(\triangle)\},$$
$$\text{where } \{\triangle', \triangle''\} := \texttt{children}(\triangle).$$

1.2.3 Finding $\mathcal{T}_{N+1}$ from $\mathcal{T}_N$

Define $\tilde{e}(\triangle)$ as in (1.2). These modified errors give insight in the quality of approximation on a leaf $\triangle \in \texttt{leaves}(\mathcal{T}_N)$. As we are now monitoring the *entire* tree, we would like some analogous measure for this entire tree.

Definition 1.16. The $\tilde{e}^{hp}_j$ is (analogously to (1.2)) defined as

$$\tilde{e}^{hp}_1(\triangle) = \tilde{e}(\triangle), \quad \tilde{e}^{hp}_j(\triangle) := \begin{cases} \left(\frac{1}{e^{hp}_j(\triangle)} + \frac{1}{\tilde{e}^{hp}_{j-1}(\triangle)}\right)^{-1} & \min\{e^{hp}_j(\triangle), \tilde{e}^{hp}_{j-1}(\triangle)\} > 0 \\ 0 & \text{else.} \end{cases}$$

To finally select the node to subdivide, the following two functions will help.

Definition 1.17. The functions $q : \mathcal{T}_N \to \mathbb{R}_{\geq 0}$ and $t : \mathcal{T}_N \to \texttt{leaves}(\mathcal{T}_N)$ are recursively defined as follows. For $\triangle \in \texttt{leaves}(\mathcal{T}_N)$,

$$q(\triangle) := \tilde{e}^{hp}_1(\triangle), \quad t(\triangle) := \triangle,$$

and for $\triangle \in \texttt{inners}(\mathcal{T}_N)$ with $\{\triangle', \triangle''\} := \texttt{children}(\triangle)$,

$$q(\triangle) := \min\left\{\max\{q(\triangle'), q(\triangle'')\}, \tilde{e}^{hp}_{r(\triangle,\mathcal{T}_N)}(\triangle)\right\}, \quad t(\triangle) := t(\operatorname{argmax}\{q(\triangle'), q(\triangle'')\}.$$

For some node $\triangle$, $q(\triangle)$ determines the best-obtainable modified *hp*-error over the set of nodes in $\mathcal{T}_{N,\triangle}$. Then, $t(\triangle)$ is the leaf in $\texttt{leaves}(\mathcal{T}_{N,\triangle})$ that is predicted to achieve the highest error reduction when subdividing. In other words, $t(D)$ is the node that we want to subdivide to get $\mathcal{T}_{N+1}$.

The final algorithm now takes a very simple form.

Algorithm 1.18 (Binev2014[7])**.** *Take N_{max} to be some maximal value. Set $T_1 = D$. Then, for $1 \leq N < N_{max}$, subdivide $t(D)$ in $\mathcal{T}_N$ to form $\mathcal{T}_{N+1}$ and set $N := N + 1$.*

Lemma 1.19 ([7, Lem. 3.2])**.** *The precise complexity of Algorithm 1.18 to obtain $(T_N, \mathcal{T}_N)$ is*

$$\mathcal{O}\left(\sum_{\triangle \in \mathcal{T}_N} r(\triangle, \mathcal{T}_N)\right).$$

Theorem 1.20 ([7, Thm. 1.2])**.** *For each N, Algorithm 1.18 defines an h-tree $\mathcal{T}_N$ and an hp-tree T_N such that the total error $E(T_N, \mathcal{T}_N)$ satisfies:*

$$n \leq N \implies E(T_N, \mathcal{T}_N) \leq \frac{2N-1}{N+1-n} E^{hp}_n.$$

Furthermore, this algorithm obtains $(T_N, \mathcal{T}_N)$ with complexity between $\mathcal{O}(N \log N)$ and $\mathcal{O}(N^2)$.

Proof. For a complete proof of the Theorem (of which the theory is beyond the scope of this thesis), see [7]. We will only prove the last assertion, using the result of Lemma 1.19, for which one only has to cover the most extreme cases.

First, consider $\mathcal{T}_N$ to be a perfectly balanced tree, i.e., having all leaves of equal depth (thus implying that N is a power of two). For each depth level d fixed, there are 2^d nodes $\triangle$ with depth d, all with degree $r(\triangle, \mathcal{T}_N) = N/2^d$. Hence, at every depth level, the total sum of degrees equals N. There are $\log_2(N)$ such levels, hence $\sum_{\triangle \in \mathcal{T}_N} r(\triangle, \mathcal{T}_N) = N \log_2 N$.

Now consider $\mathcal{T}_N$ to be perfectly *un*balanced, in the sense that there are exactly two nodes with equal depth, located at the finest level. Then, for each level d fixed, there is one node with $r(\triangle, \mathcal{T}_N) = 1$ and one with $r(\triangle, \mathcal{T}_N) = N - d$. There are N such levels, so $\sum_{\triangle \in \mathcal{T}_N} r(\triangle, \mathcal{T}_N) = \sum_{d=0}^{N-1}(1 + N - d) = N(N+3)/2$, which is $\mathcal{O}(N^2)$. □

1.2.4 Statement of Algorithm 1.18

While the above is enough to fully define the algorithm, a very precise statement was given in [7].

Algorithm 1.21 (Binev2014[7])**.** *Set N_{max} to some maximal value. We iterate as follows:*

```
1. Set $N := 1$, $\mathcal{T}_1 := D$, $r(D) := 1$, $\tilde{e}(D) := e_1(D)$, $e_1^{hp}(D) := e_1(D)$, $\tilde{e}_1^{hp}(D) := \tilde{e}(D)$,
   $q(D) := \tilde{e}(D)$ and $t(D) := D$;
2. Set $\triangle_N := t(D)$, define $\mathcal{T}_{N+1}$ by subdividing $\triangle_N$;
3. For $\triangle \in$ children$(\triangle_N)$, set $r(\triangle) := 1$, $\tilde{e}(\triangle) := \left(\frac{1}{e_1(\triangle)} + \frac{1}{\tilde{e}(\triangle_N)}\right)^{-1}$, $e_1^{hp}(\triangle) := e_1(\triangle)$,
   $\tilde{e}_1^{hp}(\triangle) := \tilde{e}(\triangle)$, $q(\triangle) := \tilde{e}(\triangle)$ and $t(\triangle) := \triangle$;
4. Set $\triangle := \triangle_N$;
5. Set $N := N + 1$, and if $N \geq N_{max}$, exit;
6. iterate:
    (a) Set $r(\triangle) := r(\triangle) + 1$ and calculate $e_{r(\triangle)}(\triangle)$;
    (b) Set $\{\triangle', \triangle''\} :=$ children$(\triangle)$;
    (c) Set $e_{r(\triangle)}^{hp}(\triangle) := \min\{e_{r(\triangle')}^{hp}(\triangle') + e_{r(\triangle'')}^{hp}(\triangle''), e_{r(\triangle)}(\triangle)\}$;
    (d) Set $\tilde{e}_{r(\triangle)}^{hp}(\triangle) := \left(\frac{1}{e_{r(\triangle)}^{hp}(\triangle)} + \frac{1}{e_{r(\triangle)-1}^{hp}(\triangle)}\right)^{-1}$;
    (e) Set $X := \operatorname{argmax}\{q(\triangle'), q(\triangle'')\}$, $q(\triangle) := \min\{q(X), \tilde{e}_{r(\triangle)}^{hp}(\triangle)\}$ and $t(\triangle) := t(X)$;
    (f) If $\triangle = D$, goto Step 2. Else, set $\triangle$ to its parent.
7. goto Step 2.
```

2 Two dimensions

Of course, most real-life problems do not reside in one dimension. Multidimensional approximation is a thriving field of research. To keep things manageable (and, of course, visualisable!), we constrain ourselves to solving two-dimensional approximation by means of tree generation.

We will shortly see that a lot of changes will have to be made. The tree-generating algorithms were presented in a dimension-invariant fashion, except for the following:

1. Generalizations of the interval to 2 dimensions: we will address rectangular and triangular elements;
2. Subdivision of these elements is far less trivial than in one dimension, so we will shed some light on the Newest Vertex Bisection and conforming partitions;
3. Polynomial approximation is again far less trivial, so we will shed light on the multidimensional generalization of Legendre polynomials, namely PKD polynomials. Furthermore, we will construct the r-dimensional function spaces defined in the introduction of Chapter 1.

2.1 Domain choice

In one dimension, our domain of choice was an interval. In two dimensions, intervals generalize to a plethora of objects. In most real-life applications, *polygons* are used.

2.1.1 Polygons

Definition 2.1. A *polygon* is a two-dimensional object that is bounded by a finite list of vertices (or, equivalently, edges), closed in a loop. A polygon is called *simple* if non-adjacent edges have empty intersection. A polygon is called *convex* if its interior I is a convex set, i.e., for every two $x, y \in I$ and $\lambda \in [0,1]$ it holds that $\lambda x + (1-\lambda)y \in I$.

As polygons are potentially very complex objects, we would like to partition it into elements that are easier the maintain.

Definition 2.2. A *polygonal triangulation* is the decomposition of a simple polygon D into a set of triangles, i.e., finding a set of triangles with pairwise non-intersecting interiors whose union is D.

There are many algorithms designed to find a polygonal triangulation. One of the easiest is the so-called ear-clipping method.

Definition 2.3. A *reflex vertex* is a vertex v_i for which the interior angle of the triangle (v_{i-1}, v_i, v_{i+1}) is less than $\pi/2$.

We will make use of the fact that every polygon P with $n \geq 4$ has at least one *ear*[14]: a triangle $(v_{i-1}, v_i, v_{i+1}) \subset P$ containing no other points of the polygon. *Clipping* this ear, i.e., removing the triangle from the polygon, yields either a triangle or a polygon with $n-1$ points which in turn must have another ear.

Algorithm 2.4 (Ear-clipping triangulation[14])**.** *Given is a counter-clockwise oriented doubly-linked list of vertices of length n. We will construct a triangulation with $n-2$ triangles.*

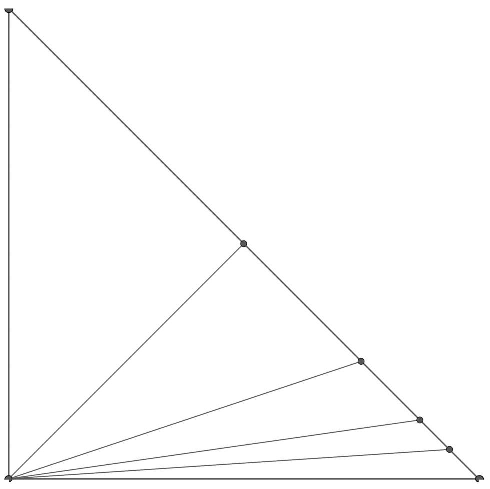


Figure 2.1: A partition with very sharp triangles.

1. *Iterate through all elements v_i of the list until the list has 3 elements left:*
    (a) *If v_i is a reflex vertex and if all vertices v_j, $j \notin \{i-1, i, i+1\}$ are not contained in the triangle $(v_{i-1}, v_i, v_{i+1}) =: \triangle$, add $\triangle$ to the triangulation and remove v_i from the list;*
    (b) *Else, continue;*
2. *Add (v_0, v_1, v_2) to the triangulation.*

Remark 2.5. While in this thesis we chose the Ear-Clipping method because it was easy, there exist methods with far better properties. One of the most widely used triangulation algorithms is the Delaunay Triangulation. The most applicable property of this method is that it finds the triangulation with *largest minimal angle*: compared to any other triangulation of the points, the smallest angle present in the Delaunay triangulation is at least as large as the smallest angle in any other[11]. To keep the thesis somewhat bounded, we will not touch on this any further.

2.1.2 Triangles

Whereas in one dimension, the natural subdivision rule is splitting the interval in two parts of equal length, no such equivalent exists in two dimensions using triangular elements.

Example 2.6. Say for instance, one would use the following subdivision rule:

> Subdivide $\triangle$ by connecting the vertex of the triangle that is in the most bottom-left position to the middle point of the opposing edge.

Of course, vertices cannot overlap, so this is a deterministic way of choosing a unique point for every triangle.

After a few subdivisions, the partition might look as in Figure 2.1. The shape of the resulting triangles is such that the lower left angle converges to 0, a very undesirable trait[1].

There is however a subdivision rule for which one can prove that subsequent subdivisions will not generate triangles with increasingly smaller smallest angles: the Newest Vertex Bisection.

[1]It was found in [17, p. 327] that the condition number of the *stiffness matrix* – one of the key matrices in the Finite Element Method – grows like $1/\sin(\theta)$, where θ is the smallest angle of $\triangle$.

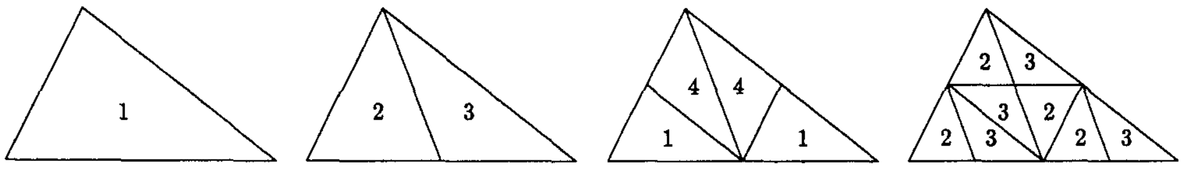


Figure 2.2: The four similarity classes of triangles obtained by Newest Vertex Bisection. [17]

2.2 Newest Vertex Bisection

The method of Newest Vertex Bisection works as follows.

> For the initial triangle $\triangle = D$, mark one of the vertices as the *newest* vertex, denoted by $v(\triangle)$. Subdivision of an arbitrary node $\triangle \in \texttt{leaves}(T)$ now happens by connecting $v(\triangle)$ with the midpoint of the opposing edge, creating a new vertex v'. For both children $\triangle', \triangle''$ of $\triangle$, we set $v(\triangle') = v(\triangle'') = v'$.

The following property of NVB is one of the main reasons why this method is so widely used.

Theorem 2.7 ([21])**.** *Using Newest Vertex Bisection, every triangle in the partition tree T is in one of four similarity classes, thus assuring no angle converges to 0: see Figure 2.2.*

2.3 Polynomial approximation on a triangle

In one dimension, we looked at multiple polynomial bases and concluded that *orthogonal bases* are well-conditioned. We'd like to extend this notion to two dimensions, but first need to define what constitutes a polynomial. Assume $\triangle \subset \mathbb{R}^2$ to be a triangular domain.

Definition 2.8. A two-dimensional mononomial $m : \triangle \to \mathbb{R}$ is the product of powers of x and y with nonnegative integer exponents:

$$m(x, y) = x^{r-k} y^k,$$

and its *degree* is equal to r.

A two-dimensional polynomial is a linear combination of two-dimensional monomials, with degree equal to the maximal degree of its terms. We will denote the set of two-dimensional polynomials by $\mathcal{P}_2(\triangle)$.

Lemma 2.9. *For every r, the set of monomials of degree up to r is a basis for $\mathcal{P}_2^r(\triangle)$, the set of polynomials with degree up to r. This basis has $(r + 2)(r + 1)/2$ elements and as such, is a space of dimension $(r + 2)(r + 1)/2$.*

Notation 2.10. We will write $m_{j,k}$ for the monomial $x^j y^k$, and m_j for $m_{\pi^{-1}(j)}$, where π is the Cantor pairing function defined in §0.2. More generally, given basis Φ, we will write $\varphi_{j,k}$ for the element of Φ with highest term $x^j y^k$, and φ_j for $\varphi_{\pi^{-1}(j)}$.

With the monomial basis in hand, two valid questions arise:

1. How is this basis conditioned, i.e., what is its condition number?

2. Are there bases with smaller condition number?

These questions are considerably harder to answer than in the previous.

2.3.1 Condition number of a two-dimensional basis

Whereas in the one-dimensional case, the ordering of the monomial basis was very explicit, we are now left with a rather implicit ordering. The following result will help resolve this apparent issue.

Lemma 2.11. *The condition number of a mass matrix is invariant under re-ordering.*

Proof. Changing the order of the basis elements amounts to a basis transformation, and using Theorem 0.35 together with the fact that Eigenvalues are invariant under orthogonal change of basis (a result easily confirmed), we arrive at the conclusion. □

Another result implicitly derived in the one-dimensional case severely helps our case.

Lemma 2.12. *The condition number of the mass matrix is invariant under domain transformations.*

Proof. Consider two triangular domains $\triangle, \tilde{\triangle} \subset \mathbb{R}^2$, with an affine map $G : \tilde{\triangle} \to \triangle$ between them. Let $\{\varphi_j : j \leq n\}$ be a basis over $\triangle$. Then by the substitution rule,

$$\begin{aligned}
(M^{\triangle})_{j,k} &= \iint\limits_{\triangle} \varphi_j(x,y)\varphi_k(x,y)dxdy \\
&= \iint\limits_{\tilde{\triangle}} \varphi_j(G(\tilde{x},\tilde{y}))\varphi_k(G(\tilde{x},\tilde{y})) \cdot |\det{(DG)(\tilde{x},\tilde{y})}| d\tilde{x}d\tilde{y} \\
&= \frac{\text{vol}(\triangle)}{\text{vol}(\tilde{\triangle})} \iint\limits_{\tilde{\triangle}} \varphi_j(G(\tilde{x},\tilde{y}))\varphi_k(G(\tilde{x},\tilde{y}))d\tilde{x}d\tilde{y} \\
&= \frac{\text{vol}(\triangle)}{\text{vol}(\tilde{\triangle})} (M^{\tilde{\triangle}})_{j,k},
\end{aligned}$$

so every element $(M^{\tilde{\triangle}})_{j,k}$ is a constant multiple of $(M^{\triangle})_{j,k}$. Together with Theorem 0.35, the conclusion must hold. □

With this last result, we can now resort to finding the mass matrix by considering a *reference triangle* $\hat{\triangle}$ only. See Figure 2.3 for the reference triangle we will be using.

We conclude with the following result found in literature.

Theorem 2.13 ([9])**.** *The condition number of the monomial basis $\mathcal{M}_n$ is exponential in n.*

Proof. While not technically a proof, a numerical verification of this result can be found in Appendix B. □

2.3.2 Proriol-Koornwinder-Dubiner orthogonal polynomial basis

Back in 1991, Dubiner [13] faced this exact problem. Using techniques developed by Koornwinder [15] and Proriol [18], he constructed a set of orthogonal polynomials in a *very* elegant manner. Unfortunately, the theory provided in [13] and its formulation is beyond the scope of this thesis.

The basis proposed in the article mentioned makes use of both Legendre polynomials and a generalization thereof, namely *Jacobi polynomials*. We will present a slightly simplified version.

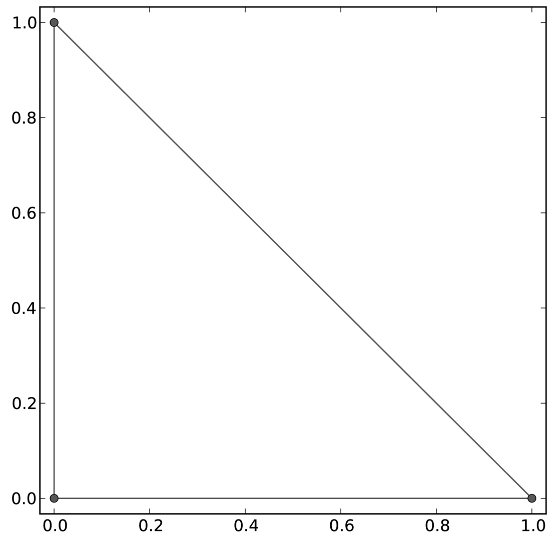


Figure 2.3: The reference triangle $\hat{\triangle}$.

Definition 2.14. For integer $\alpha > -1, j \geq 0$, the Jacobi polynomial $P_j^{(\alpha,0)}(x)$ is defined on $x \in [-1,1]$, and has the following property (orthogonality for the weight function $(1-x)^\alpha$):

$$\int_{-1}^{1} P_j^{(\alpha,0)}(x) P_k^{(\alpha,0)}(x)(1-x)^\alpha dx = \begin{cases} \frac{2^{\alpha+1}}{2j+\alpha+1} & \text{if } j = k; \\ 0 & \text{else.} \end{cases}$$

If $\alpha = 0$, this polynomial coincides with the Legendre polynomial. The *shifted Jacobi polynomial* is constructed by the analogue of (5).

To evaluate a Jacobi polynomial in a point x, the following is useful.

Theorem 2.15 ([2, (22.1.4)])**.** *The following 3-term recurrence relation holds:*

$$\begin{aligned} P_0^{(\alpha,0)}(x) &= 1, \\ P_1^{(\alpha,0)}(x) &= ((\alpha+2)x + \alpha)/2, \\ 2j(j+\alpha)(2j+\alpha-2)P_j^{(\alpha,0)}(x) &= (2j+\alpha-1)\{(2j+\alpha)(2j+\alpha-2)x + \alpha^2\}P_{j-1}^{(\alpha,0)}(x) \\ &\quad - 2(j+\alpha-1)(j-1)(2j+\alpha)P_{j-2}^{(\alpha,0)}(x). \end{aligned}$$

Using these Jacobi polynomials, we can define the following family of functions.

Definition 2.16 ([13, (5.4)], [9, (1.1)], [26])**.** The Proriol-Koornwinder-Dubiner (PKD) polynomials $Q_{j,k}$ are two-dimensional polynomials on the reference triangle $\hat{\triangle}$, defined as:

$$Q_{j,k}(x,y) := P_j\left(\frac{2x}{1-y} - 1\right) \cdot (1-y)^j \cdot P_k^{(2j+1,0)}(2y-1).$$

Remark 2.17. While this looks somewhat terrifying, it is merely the polynomial

$$Q_{j,k}^{\square}(s,t) := P_j(s)\left[\frac{1-t}{2}\right]^j P_k^{(2j+1,0)}(t), \quad (s,t) \in [-1,1]^2 =: \square \tag{2.1}$$

transformed to the reference triangle by the function

$$\begin{cases} F : \square \to \hat{\triangle} : (s,t) \mapsto ((1-t)(1+s)/4, (t+1)/2) \\ F^{-1} : \hat{\triangle} \to \square : (x,y) \mapsto (2x/(1-y) - 1, 2y-1). \end{cases}$$

This function has Jacobian $(1-t)/8$.

Theorem 2.18. *The PKD polynomials have the following property (orthogonality on the reference triangle):*

$$\iint\limits_{\hat{\triangle}} Q_{j,k}(x,y)Q_{l,m}(x,y)dxdy = \frac{\delta_{j,l}\delta_{k,m}}{2(2j+1)(j+k+1)}.$$

Proof. We can split the integral:

$$\begin{aligned}\iint\limits_{\hat{\triangle}} Q_{j,k}(x,y)Q_{l,m}(x,y)dydx &= \iint\limits_{\square} P_j(s)P_l(s)\left[\frac{1-t}{2}\right]^{j+l} P_k^{(2j+1,0)}(t)P_m^{(2l+1,0)}(t)|DF(s,t)|dtds \\ &= \frac{1}{4}\iint\limits_{\square} P_j(s)P_l(s)\left[\frac{1-t}{2}\right]^{j+l+1} P_k^{(2j+1,0)}(t)P_m^{(2l+1,0)}(t)dtds \\ &= \frac{1}{4}\int_{-1}^{1} P_j(s)P_l(s)ds \int_{-1}^{1}\left[\frac{1-t}{2}\right]^{j+l+1} P_k^{(2j+1,0)}(t)P_m^{(2l+1,0)}(t)dt.\end{aligned}$$

This enables us to use Theorem 0.34 to reduce this to

$$\frac{1}{4}\delta_{j,l}\frac{2}{2j+1}\int_{-1}^{1}\left[\frac{1-t}{2}\right]^{j+l+1} P_k^{(2j+1,0)}(t)P_m^{(2l+1,0)}(t)dt.$$

This enables us to set $l = j$ in the remaining integral, to find

$$\int_{-1}^{1}\left[\frac{1-t}{2}\right]^{j+l+1} P_k^{(2j+1,0)}(t)P_m^{(2l+1,0)}(t)dt = \int_{-1}^{1}(1-t)^{2j+1}2^{-2j-1}P_k^{(2j+1,0)}(t)P_m^{(2j+1,0)}(t)dt,$$

which can be reduced using Definition 2.14 to

$$\delta_{k,m}2^{-2j-1}\frac{2^{2j+2}}{2j+2k+2} = 2[2j+2k+2]^{-1}.$$

The result is now clearly true. □

Theorem 2.19. *The set of PKD polynomials up to degree r – $\{Q_{j,k} : 0 \le j, k; j+k \le r\}$ – is an orthogonal basis for $\mathcal{P}_2^r(\hat{\triangle})$.*

Proof. Orthogonality is ensured by the above Theorem. To show it is a basis, we want to show that the degree of $Q_{j,k}$ is indeed r or less, and that it spans the whole space.

Note that $Q_{j,k}$ is the product of $P_j(2x/(1-y)-1)$, $(1-y)^j$ and $P_k^{(2j+1,0)}(2y-1)$. Now, this first factor is a polynomial of degree j in $2x/(1-y)-1$, so we can write

$$\begin{aligned}P_j(2x/(1-y)-1)(1-y)^j &= \sum_{k=0}^{j} a_k(2x/(1-y)-1)^k(1-y)^j \\ &= \sum_{k=0}^{j} a_k\left[\sum_{l=0}^{k}\binom{k}{l}2^{k-l}\left(\frac{x}{1-y}\right)^{k-l}(1-y)^j(-1)^l\right] \\ &= \sum_{k=0}^{j} a_k\left[\sum_{l=0}^{k}\binom{k}{l}(2x)^{k-l}(1-y)^{l-k+j}(-1)^l\right],\end{aligned}$$

so that every term is a bivariate polynomial of degree $k-l$ in x and $l-k+j$ in y, for a total degree of $k-l+l-k+j = j$. Hence, the first two factors of $Q_{j,k}$ combined is a polynomial in

x and y of degree j. Of course, the last factor is a polynomial of degree k in y. Their product therefore has degree $j + k \leq r$. We conclude that $Q_{j,k} \in \mathcal{P}_2^r$ for every j, k.

Furthermore, this mutually orthogonal set contains no zero polynomial, so they must all be linearly independent. The set has $(r + 2)(r + 1)/2$ elements – exactly the same as the standard basis for $\mathcal{P}_2^r$. We conclude that it must be a basis. $\square$

We conclude this argument with the answer to the original question.

Theorem 2.20 ([9, Thm. 3.1])**.** *The condition number of the PKD basis, after normalization, is* 1.

2.3.3 Extending to other triangles

We will now extend our view to other triangles. Assume $\triangle$ to be the triangle with vertices

$$(x_1, y_1), (x_2, y_2), (x_3, y_3).$$

Definition 2.21. The area or volume of $\triangle$ is defined as

$$\operatorname{vol}(\triangle) := \tfrac{1}{2} \left| -x_2 y_1 + x_3 y_1 + x_1 y_2 - x_3 y_2 - x_1 y_3 + x_2 y_3 \right|.$$

Definition 2.22. A triangle is called *degenerate* if its vertices are collinear, or equivalently, has volume 0. In the following, all triangles will be assumed nondegenerate.

Lemma 2.23. *Given a triangle $\triangle$, consider the affine map G defined by*

$$G : \hat{\triangle} \to \triangle : (x, y) \mapsto \begin{bmatrix} x_2 - x_1 & x_3 - x_1 \\ y_2 - y_1 & y_3 - y_1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} x_1 \\ y_1 \end{bmatrix}.$$

This is a bijection between $\hat{\triangle}$ and $\triangle$, with Jacobian

$$|DG(x, y)| = \frac{\operatorname{vol}(\triangle)}{\operatorname{vol}(\hat{\triangle})} = 2 \operatorname{vol}(\triangle),$$

and inverse

$$G^{-1}(x, y) = \frac{1}{2 \operatorname{vol}(\triangle)} \begin{bmatrix} y_3 - y_1 & x_1 - x_3 \\ y_1 - y_2 & x_2 - x_1 \end{bmatrix} \left(\begin{bmatrix} x \\ y \end{bmatrix} - \begin{bmatrix} x_1 \\ y_1 \end{bmatrix} \right).$$

Definition 2.24. For a triangle $\triangle$, the *shifted PKD polynomials* $Q_{j,k}^{\triangle}(x, y)$ are defined as follows:

$$Q_{j,k}^{\triangle}(x, y) := Q_{j,k}(G^{-1}(x, y)).$$

Theorem 2.25. *For a triangle $\triangle$, the following equality holds for the shifted PKD polynomials:*

$$\iint\limits_{\triangle} Q_{j,k}^{\triangle}(x, y) Q_{l,m}^{\triangle}(x, y) dx dy = \frac{\delta_{j,l} \delta_{k,m} \operatorname{vol}(\triangle)}{(2j + 1)(j + k + 1)}.$$

Proof. This is a simple application of the substitution rule. $\square$

2.3.4 Choosing V_r

In one dimension, the space of polynomial of degree $r-1$ has dimension r. In two dimensions however, this space is of dimension $r(r+1)/2$. For the tree generating algorithms, we desire an r-dimensional space V_r as mentioned in the introduction to Chapter 1. A first idea might be to take the first r basis elements of some 2 dimensional polynomial basis (such as the PKD-basis defined in Theorem 2.19). This creates an asymmetry: if we look at the first two elements of the monomial basis – 1 and x – with which we construct polynomials of best approximation, a higher degree in the x-axis is present than in the y-axis. Moreover, in general, the local error of approximation is thought[2] to not improve much over an approximation with one degree of freedom.

Recall that if $r = (n+2)(n+1)/2$ for some $n \geq 0$, then approximation using the first r basis elements *does* yield a symmetric construction as we are now approximating with the polynomials of degree n. We create a function space V_r that *only* considers such 'symmetric' r:

$$V_r := \mathcal{P}_2^{n(r)}, \quad n(r) := \max\{n \geq 0 : (n+2)(n+1)/2 \leq r\}.$$

In other words, we take the largest n for which $(n+2)(n+1)/2$ is not larger than r, and approximate with polynomials of degree n.

Remark 2.26. An immediate consquence of this is that

$$e_1(\triangle) = e_2(\triangle); \quad e_3(\triangle) = e_4(\triangle) = e_5(\triangle); e_6(\triangle) = e_7(\triangle) = e_8(\triangle) = e_9(\triangle); \ldots,$$

as $V_1 = V_2$, $V_3 = V_4 = V_5$, ...

2.3.5 Polynomial of best approximation

As in the one-dimensional case, we can state the approximation problem.

$$\text{Given triangle } \triangle,\ f \in L^2(\triangle) \text{ and } r \geq 1, \text{ find } p_{n(r)} \in \mathcal{P}_2^{n(r)}(\triangle) \text{ such that}$$
$$e_r(\triangle) = \|f - p_{n(r)}\|_{2,\triangle}. \tag{2.2}$$

Lemma 2.27. *There is a unique polynomial that solves* (2.2).

Proof. As $\mathcal{P}_2^{n(r)}(\triangle)$ is non-empty, closed and convex, Theorem 0.4 applies. □

Theorem 2.28. *Given $r \geq 1$, given triangle $\triangle$ and $f \in L^2(\triangle)$, and an orthogonal basis*

$$\{\varphi_{j,k} : 0 \leq j, k; 0 \leq j + k \leq n(r)$$

for $\mathcal{P}_2^{n(r)}(\triangle)$, the polynomial

$$p_{n(r)}(x,y) := \sum_{j=0}^{n(r)} \sum_{k=0}^{n(r)-j} \gamma_{j,k}\varphi_{j,k}(x,y), \quad \gamma_{j,k} := \frac{\langle f, \varphi_{j,k}\rangle}{\langle \varphi_{j,k}, \varphi_{j,k}\rangle} \tag{2.3}$$

is the polynomial of best approximation, with inner products taken over $\triangle$.

[2] A strong argument for this is the following. Consider some smooth bivariate f and look at its Taylor expansion around (for instance) 0, with x, y in some ball of radius $h \ll 1$:

$$f(x,y) = f(0,0) + [xf_x(0,0) + yf_y(0,0)] + \frac{1}{2!}[x^2 f_{xx}(0,0) + 2xyf_{xy}(0,0) + y^2 f_{yy}(0,0)] + \cdots, x^2 + y^2 \leq h^2.$$

If we approximate f by a polynomial using the first 2 basis elements, we can "approximately" eliminate the first 2 terms of the Taylor expansion. As the order of magnitude of the approximation error is governed by the first term of the remaining Taylor expansion, the approximation error is still of order h. Approximation using the first *three* basis elements however yields an approximation error of $\mathcal{O}(h^2)$ which *is* an improvement.

Proof. The space $\mathcal{P}_2^{n(r)}(\triangle)$ is finite-dimensional and hence closed. Now use Theorem 0.5. $\square$

Theorem 2.29. *Given the same prerequisites as in Theorem 2.28, the following holds:*

$$\|f - p_{n(r)}\|_{2,\triangle}^2 = \|f\|_{2,\triangle}^2 - \|p_{n(r)}\|_{2,\triangle}^2 = \|f\|_{2,\triangle}^2 - \sum_{j=0}^{n(r)} \sum_{k=0}^{n(r)-j} \gamma_{j,k}^2 \langle \varphi_{j,k}, \varphi_{j,k} \rangle.$$

Proof. The last equality is a direct consequence of Theorem 0.6. As $\mathcal{P}_2^{n(r)}(\triangle)$ is a closed linear subspace of $L^2(\triangle)$, we invoke Theorem 0.7 for $x = f$ to find that $f = f - p_{n(r)} + p_{n(r)}$ and $\|f\|^2 = \|f - p_{n(r)}\|^2 + \|p_{n(r)}\|^2$. The result follows. $\square$

2.4 Conclusion

Almost all of the pieces of the puzzle are in order. There is one hole left to address. Given a polygonal domain and its triangulation T, one generally cannot represent this in binary tree form. We will therefore expand the definition of a tree by relaxing the requirement that a tree can only have one root. A *multiple rooted tree* is a tree with possibly more than one root, each component (being the subtree rooted at one of the roots) being a tree in the classical sense.

In the h-refining case, we can still consider all leaves of the (multiple rooted) partition tree to find which elements to subdivide, and receive near-optimal h-trees. In the hp-refining case of Algorithm 1.18, we are not able to complete Step 6 when $\triangle = D$ and D is no triangle (but for instance, a 4-sided polygon). This is because – in this thesis – we have only considered approximation over triangles. Therefore, we adapt Step 6f so the algorithm traverses up the tree until it has found one of the multiple roots:

If $\triangle$ is a root, `goto` Step 2. Else, set $\triangle$ to its parent.

This results in the following idea – we will consider a little more involved version in §5.1.

Algorithm 2.30 (Binev2014 for polygonal domains)**.** *Set N_{max} to some maximal value. Let P be the initial partition. Then, for each $\triangle \in P$, call Algorithm 1.18 with the adaptation above.*

As we are effectively dividing our problem, Algorithm 2.30 is still near-optimal. Using the different algorithms presented, we are now able to construct a triangular partition of a polygon by means of the Ear-Clipping method, and adaptively refine this partition using h- or hp-refinement. On each element of the resulting partition, we can approximate our function with the PKD-polynomials.

Part II

Practice

In this Part, we will take a look at the more practical side of the theory provided. We will first look at some results, i.e., run the (one- and two-dimensional) algorithm on some sample functions and see the performance.

Implementation

Implementation of the theory comes in a few forms. The 1D implementation can be viewed as a 'play' version to get acquainted and to find where possibilities of improvement lie. The process of writing this 1D version consisted of a few pieces of code, namely

- a test version in Python 2.7;
- a first implementation in C, based on [6];
- a second implementation in C, based on Algorithm 1.18.

One should note that in the making of this thesis, the key algorithm (Algorithm 1.18) was still in active development. The first one-dimensional implementation was made before the complete overhaul and therefore concerns the theory outlined in [6] as opposed to §1.2.

The 2D implementation and its improvements are outlined in Chapter 4. The second implementation of the one-dimensional case is based on this two-dimensional implementation and will thus receive no further attention.

3 Results and experiments in one and two dimensions

In this chapter, we will investigate the *h*- and *hp*-refinement strategies. We will start with a relatively easy example in §3.1, and compare results with literature in §3.2 and §3.3.

3.1 One dimension: a discontinuous function

Consider the function

$$f(x) := \begin{cases} e^x/e^3 & x < 3 \\ \sin(\pi x/2) & x \geq 3 \end{cases}, \quad x \in [0, 5]. \tag{3.1}$$

Plots are shown in Figures 3.1 and 3.2. The error progression can be seen in Figure 3.3.

This function is discontinuous, so we see *h*-refinement around $x = 3$. Moreover, in the *hp*-case, we see *p*-refinement on the smooth parts of f. This is exactly what we expected. We can also see that the *hp*-approximation yields better performance for given degrees of freedom. This is of course at the expense of computation time (cf. §4.1).

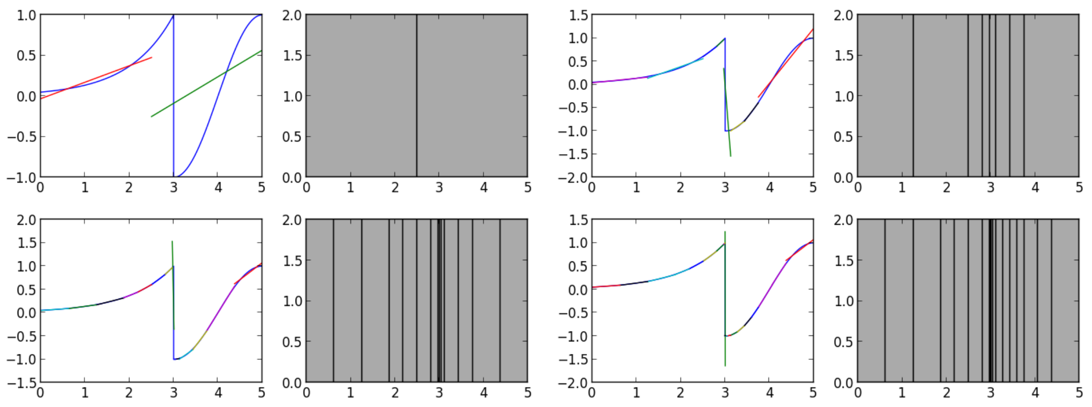


Figure 3.1: Approximation of f from (3.1) using h-refinement with 2 degrees of freedom per element. Shown: 4, 16, 28, 40 total degrees of freedom. Every figure has a plot (left) and the distribution of degrees of freedom (right).

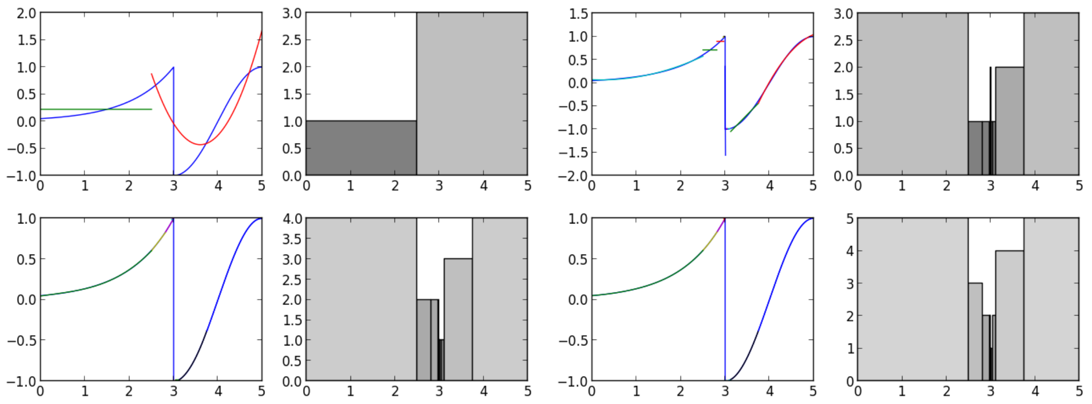


Figure 3.2: Approximation of f from (3.1) using hp-refinement. Shown: 4, 16, 28, 40 total degrees of freedom. Every figure has a plot (left) and the distribution of degrees of freedom (right).

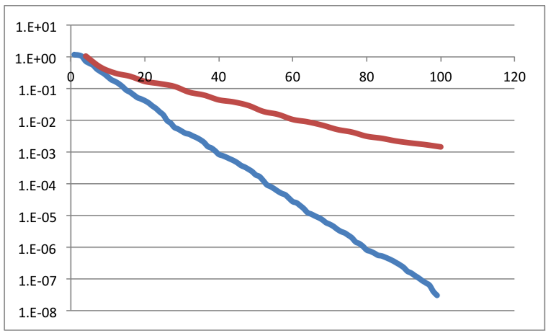


Figure 3.3: Approximation of f from (3.1) using (red) h- and (blue) hp-refinement. Shown: (x-axis) total degrees of freedom versus (y-axis) total error.

3.2 One dimension: $x^{0.7}$

In [12], an adaptive *hp*-FEM strategy is considered. The strategy described finds adaptive refinements in much the same way we have done in the previous Chapters, and thus, it makes sense to compare results. However, in [12], non-midway subdivision of an interval is allowed. It is not known whether or not this strategy was actually used in the numerical results provided.

In [12, §4.2], the function $x^{0.7}$ is approximated in *energy norm* with piecewise polynomials (the FEM approximant). The energy norm $\|f\|_{E(D)}$ of f over a domain D is defined as $\|f'\|_{2,D}$. The error considered is thus of the form

$$\|u - u_N\|_{E(D)} = \|(u - u_N)'\|_{2,D}, D = [0, 1],$$

where u_N is an approximant to u with N degrees of freedom.

Converting this to our case, if $u(x) = x^{0.7}$, then we will consider

$$g(x) = 0.7x^{-0.3}. \tag{3.2}$$

See Figure 3.4 for a plot of a sample approximation using 90 degrees of freedom. This choice of 90 is not arbitrary: this is the number of degrees of freedom Dörfler used to create the left grid of Figure 3.5, meaning we can compare it.

The resemblance between the two methods is that both error graphs have approximately the same *shape*, indicating that the convergence rate is of the same type. There are two differences however. First, our method uses polynomials of lower degree, and second, our error is slightly higher than that of [12].

It is interesting to see that in Figure 3.4, the left-most element of the partition is decorated with three degrees of freedom, whereas in Figure 3.5 this is not the case. This is likely a numerical error, as near $x = 0$, g is (almost) singular. Numerical integration libraries often do not fare well in these cases.

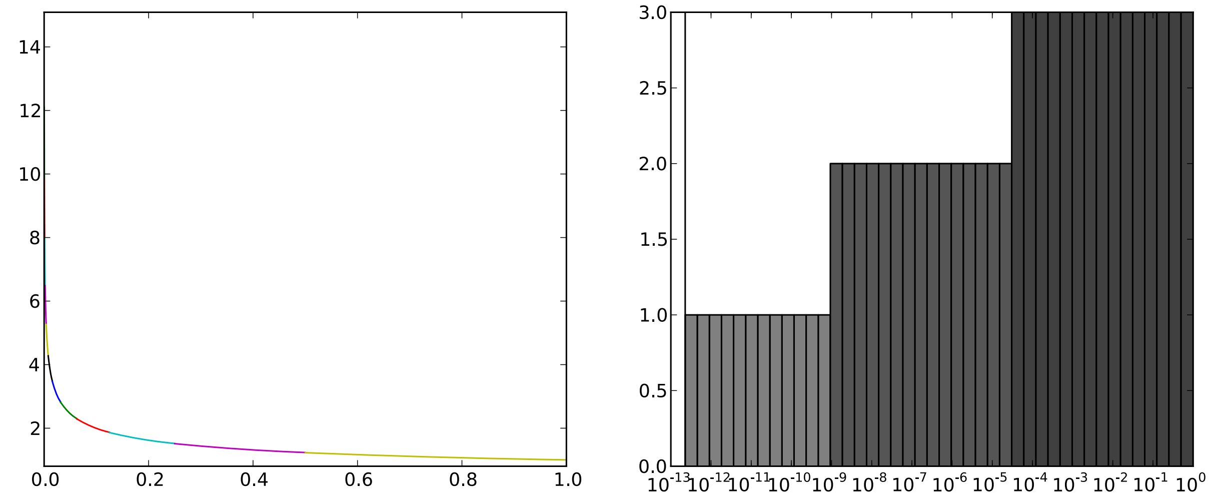


Figure 3.4: Approximation of g from (3.2) using hp-refinement with 90 degrees of freedom. Left: plot of the approximation. Right: log plot of distribution of the polynomial degree on the grid.

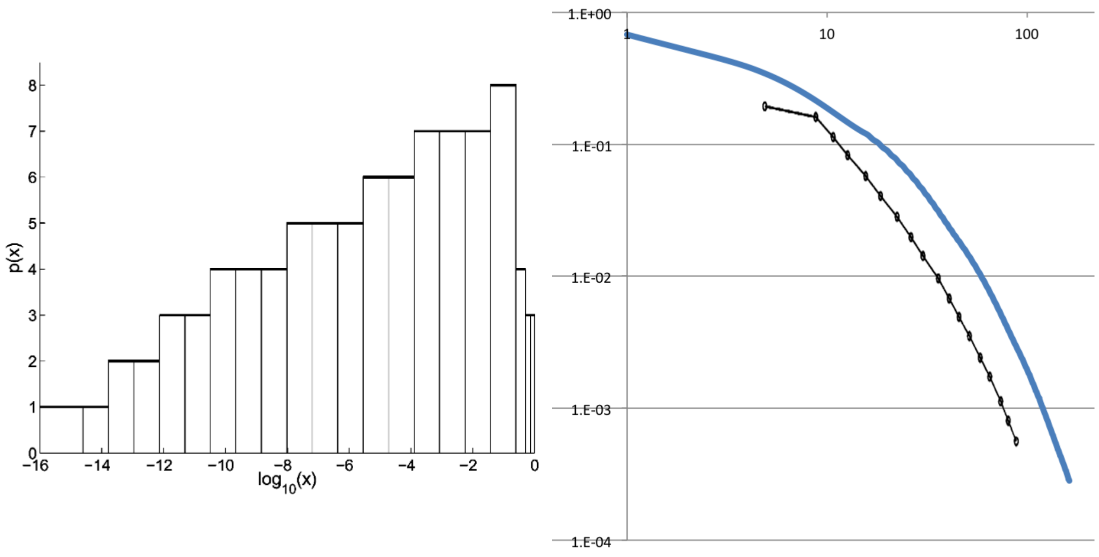


Figure 3.5: Left: a log plot of distribution of the polynomial degree on the grid of approximation of g from (3.2) using the strategy provided in [12] with 90 degrees of freedom. Right: A log-log plot showing approximation error E_N of g versus degrees of freedom N. Black thin line: results from [12]. Blue thick line: results using Algorithm 1.18.

3.3 Two dimensions: r^α

Consider the function

$$h_\alpha(x,y) := r^\alpha, \quad r := \sqrt{x^2+y^2}, \alpha \in \mathbb{R}, \tag{3.3}$$

on some disk $C_R := \{(x,y) \in \mathbb{R}^2 : r < R\}$. We require functions to be square-integrable, so that we can use Example 1.2 for the local error of approximation. Therefore, it is interesting to find out for which α, h_α is in L^2. We compute, using a transformation to polar coordinates,

$$\iint_{C_R} |h_\alpha(x,y)|^2 dydx = \int_0^R \int_0^{2\pi} |r^\alpha|^2 r d\theta dr = 2\pi \int_0^R r^{2\alpha+1} dr = 2\pi \frac{R^{2\alpha+2}}{2\alpha+2}.$$

For h_α to be L^2, this integral must be less than infinity. Therefore, if $\alpha > -1$, h_α is L^2. The value of R doesn't even matter. A consequence of this is that h_α is then L^2 on any region, such as a polygon (as every region is a subset of some disk).

For $\alpha = -1/3$, the function h_α is thought to have a (to be further specified) connection with the solution[1] to

$$\begin{cases} -\Delta u = 1 & (x,y) \in D \\ u = 0 & (x,y) \in \partial D \end{cases}, \quad D := (-1,1)^2 \setminus [-1,0]^2.$$

The *hp*-FEM approximation to this problem is numerically studied in [16, Ex. 4.6]. Again, this solution is approximated in the *energy norm*

$$\|g\|_{E(D)} := \|\nabla g\|_{2,D} = \sqrt{\iint_D (g_x)^2 + (g_y)^2 dydx}.$$

The goodness of fit of *hp*-FEM approximant u_N in N degrees of freedom is then measured as

$$\|u - u_N\|_{E(D)},$$

and the connection with our case is the following:

$$\|u - u_N\|_{E(D)} \approx \|h_{-1/3} - h^N_{-1/3}\|_{2,D},$$

where $h^N_{-1/3}$ is the *hp*-approximant to $h_{-1/3}$ found with our methods.

We will be approximating $h_{-1/3}$ on this domain. It is known[20, Thm. 4.63] that the best *hp*-FEM approximant error decays like

$$Ce^{-bN^{1/3}},$$

and hence we will make a log-plot of the error against $N^{1/3}$ and expect to see a straight line. See the left of Figure 3.6. Indeed: both the black and blue graphs exhibit a line-like decay, with equal slope. The difference in total error can stem from a plethora of reasons[2]

As [16, Ex. 4.6] provides graphs of the partitions at different steps of the *hp*-FEM approximation, it is interesting to compare it with ours. They however use rectangles as elements of the partition whereas we use triangles, and their function is different from ours along ∂D (as can be seen in the mesh refinement around the edges). See Figure 3.7.

[1]It must be noted that the exact solution to this problem is not known. Only numerical approximations exist.

[2]For one, we are approximating different functions (solution to some differential equation versus explicit function). Secondly, we are using triangles instead of rectangles like in [16]. Finally, the initial partition is somewhat unfortunate (as generated by Algorithm 2.4). See the right of Figure 3.6.

The partition formed by Algorithm 1.18 on the triangle $(0,0),(1,0),(0,1)$ is pictured in Figure 3.8. We can see a resemblance with the right partition of Figure 3.7, where the h-refinement is more prevalent closer to the singularity of $r = 0$, and p-refinement is more prevalent further away from $r = 0$.

It is somewhat hard to see, but as in the one-dimensional case of §3.2, there is p-refinement in the elements closest to 0 (where one would expect h-refinement).[3] Furthermore, in the right triangulation of Figure 3.8, we can see various elements with 4 degrees of freedom. However, Remark 2.26 holds, so 4 degrees of freedom provides no additional accuracy over 3.[4]

In the h-refining case, $h_{-1/3}$ is also interesting to consider. It is known[20] that h-refinement will converge algebraically, i.e.,

$$E_N \sim N^C, \quad C < 0,$$

where E_N is the error of approximation at N degrees of freedom. We will approximate f using 1, 3, 6 and 10 degrees of freedom per triangle. This corresponds to polynomials of degree 1, 2, 3 and 4 in two dimensions. A graph of the error can be seen in Figure 3.9. We opted for a log-log plot, because the monomials correspond with straight lines in a log-log plot. We can see that (after an initial 'bulge') the lines look straight, indicating that we indeed have algebraic convergence. Furthermore, we can see that given N, better performance is achieved for more degrees of freedom per element.

Lastly, looking at Figure 3.10, we see that partition refinements are done mainly around the singularity in $r = 0$. While in both partitions the number of triangles is the same, the biggest notable difference lies in the distribution. The right partition has a very high density around $r = 0$ while the left partition is a more global refinement. Note that this is exactly what one would expect to happen: the further away from zero, the more $h_{-1/3}$ resembles a smooth function and thus is better approximated by higher-order polynomials.

[3]This is most probably a result of poor numerical accuracy, because of the singularity at $r = 0$ (cf. §3.2).

[4]The reason for this is not entirely clear, but Algorithm 1.18 Steps 6c and 6e contain occurrences of $\min\{\}$ whose implementation always chooses the first element in case of equality (which could correspond to p-refinement).

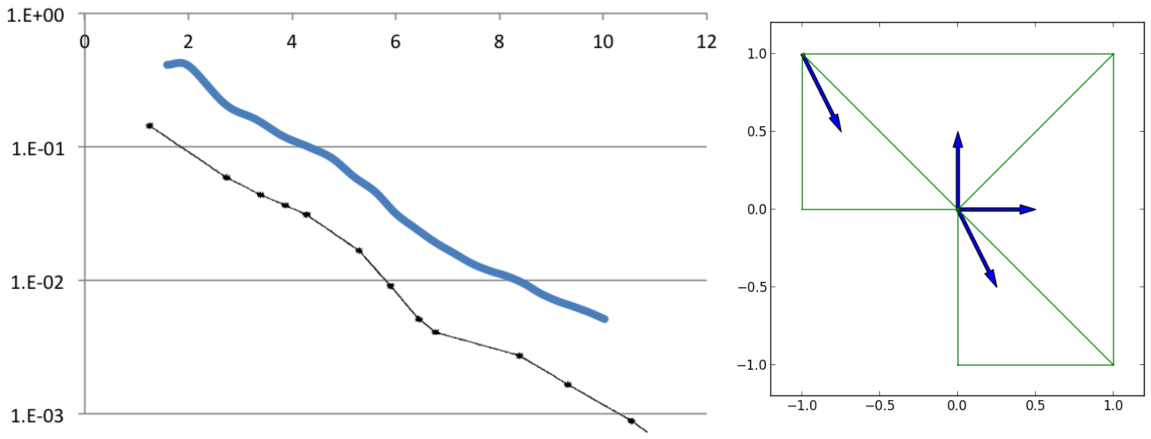


Figure 3.6: Left: Approximation error (y-axis) versus $N^{1/3}$. Blue thick graph: approximation error of $h_{-1/3}$ using hp-refinement. Black thin graph: overlay of results from [16, Ex. 4.6]. Right: initial partition of our approximation.

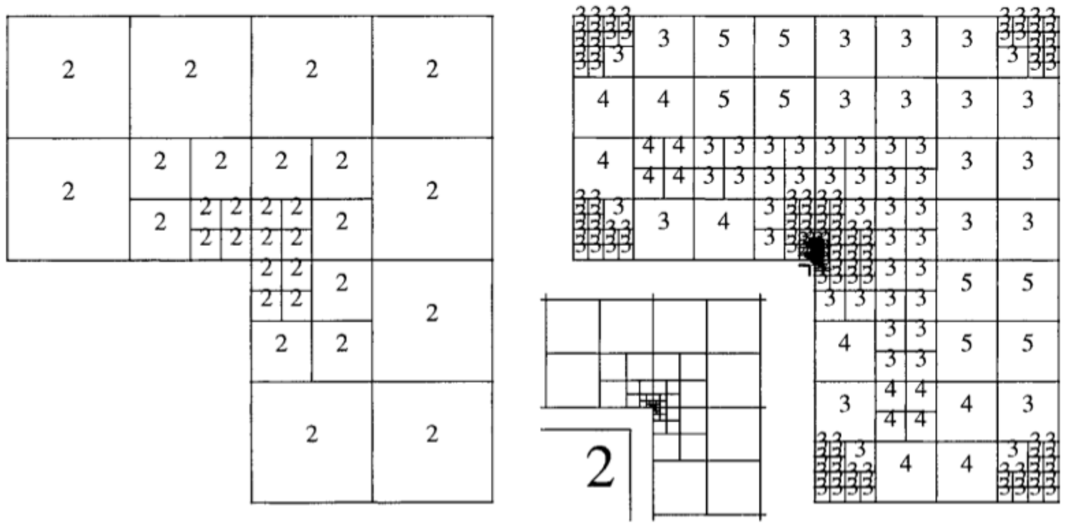


Figure 3.7: Results from [16]. Left: partition with 60 degrees of freedom. Right: partition with around 600 degrees of freedom.

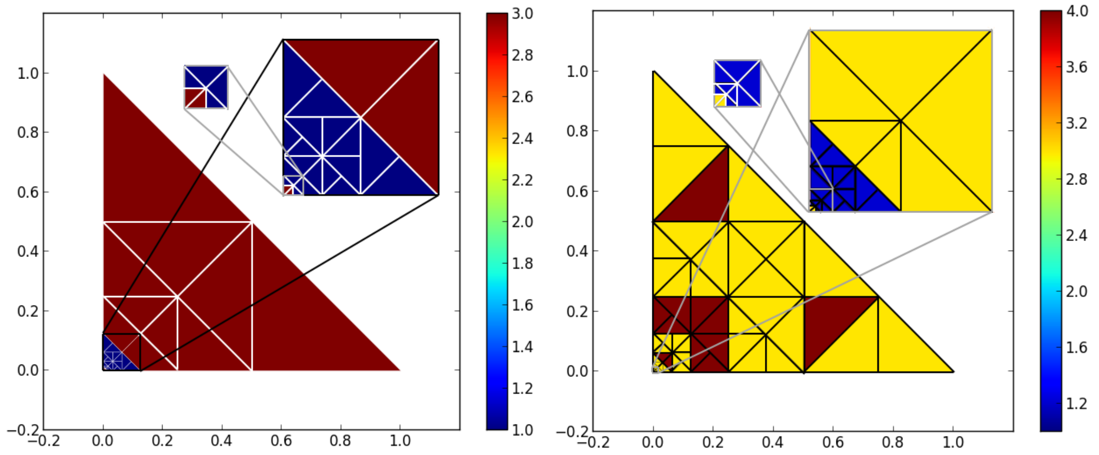


Figure 3.8: Left: partition with 60 degrees of freedom. Right: partition with 200 degrees of freedom.

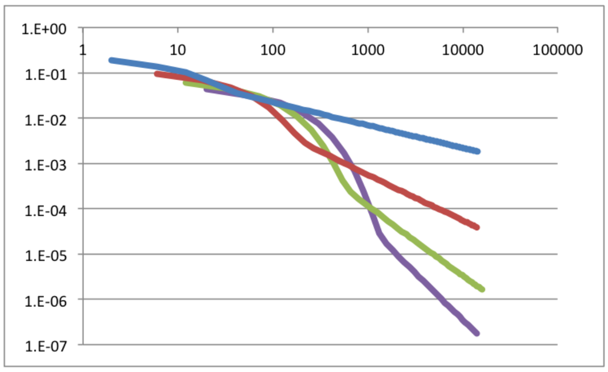


Figure 3.9: Log-log plot concerning $h_{-1/3}$ of (3.3). Plot of degrees of freedom (x-axis) versus approximation error (y-axis) with h-refinement using 1 (blue), 3 (red), 6 (green), 10 (purple) degrees of freedom per element. A straight line in a log-log plot corresponds with convergence rate of the form aN^C.

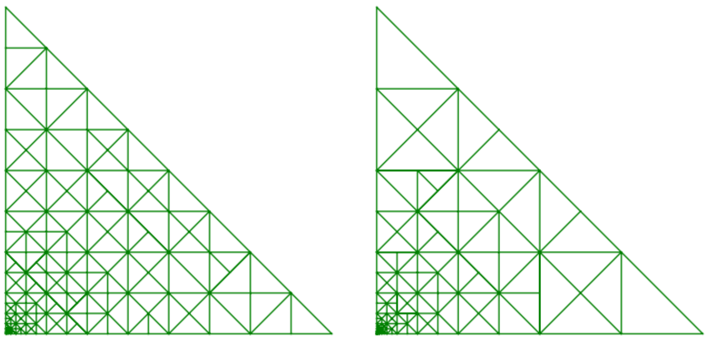


Figure 3.10: Partition of the h-approximation to $h_{-1/3}$ of (3.3) with 100 triangles, using (left) 1, (right) 10 degrees of freedom per element.

4 C-implementation in 2 dimensions

As stated before, the 2D implementation uses a slightly other version of the *hp*-adaptive algorithm than the 1D implementation. This does not mean implementation gets easier, as the transition from 1 to 2 dimensions introduces many hurdles.

The 1D implementation had a few options for improvement. First, it relied heavily on list traversal. The major downside of this choice is that list traversal is inherently sequential, making concurrency impossible where lists are used.

Secondly, the set of leaves was not explicitly available. As this list only changes once per iteration, one can easily store the set and retrieve it in constant time. Keeping this as a sorted array (as suggested in §1.1.4) makes finding which leaf to subdivide a constant-time task.

Furthermore, some problems not present in 1D arise. Most notably, given node $\triangle$, we want to be able to find neighbours with same depth (distance to root) in constant time to quicken various methods from this thesis. We refer to Appendix D for the details and usage.

All in all, this implementation is much denser than on 1 dimension and contains hardly any redundancies as-is.

The framework contains code to print the resulting tree in a pipe-friendly syntax, ready to be imported into the (albeit hardly efficient) Python code to facilitate plots.

4.1 Performance tests

We will use the function

$$h(x, y) := sin(2\pi x)cos(2\pi y)$$

on the initial triangle with vertices $(0, 0), (1, 0), (0, 1)$. Moreover, we will be using `epsabs` to be 10^{-5} (which gives results much faster at the expense of precision – but because we are merely interested in timing, combined with the fact that integration takes up virtually *all* CPU cycles, justifies the choice).

4.1.1 *h*-refinement

In Theorem 1.8, it is asserted that Algorithm 1.9 runs in linear time. See the left side of Figure 4.1. It seems as though there is a slight superlinearity[1]. A more thorough research using Kcachegrind[27] revealed that the distribution of CPU cycles ("in which functions most time is spent") is roughly the same for $N = 1000$ and $N = 10000$ with $r = 1$.

4.1.2 *hp*-refinement

In Lemma 1.20 it is stated that the complexity of Algorithm 1.18 is between $\mathcal{O}(N \log N)$ and $O(N^2)$. The right graph of Figure 4.1 resembles a parabola, thus one could conclude that (at the very least) there is no evidence stating otherwise.

It is interesting to note that the error produced by *h*-refinement on rougly 20K degrees of freedom is of order 10^{-5}, taking 20 seconds to compute. The *hp*-refining algorithm takes less

[1]An argument for this can be the following. With few degrees of freedom, everything can reside in cache. As the number of degrees of freedom increases, more cache misses arise. This can be very costly, and might very well be the reason for this superlinearity.

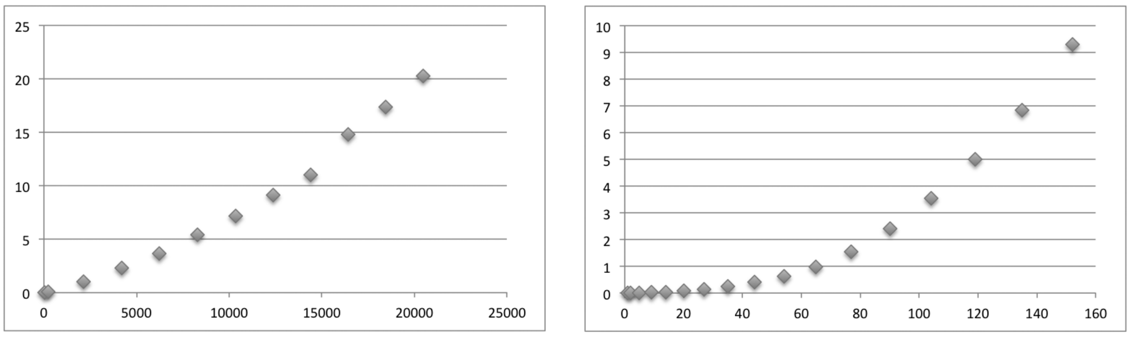


Figure 4.1: Degrees of freedom versus computation time by computing approximants to h. Left: h-refinement with $r = 1$. Right: hp-refinement.

than 10 seconds to find an approximation in less than 160 degrees of freedom with an error of order 10^{-16}.

Furthermore, it was found that more than 99% (!) of the CPU-cycles were spent on finding $e_r(\triangle)$, of which 65% computing the values of $\gamma_{j,k}$ from Theorem 2.28 and 34% computing the error, given the polynomial of best approximation. The reason for this is of course the horrendous upward recurrence of the Legendre and Jacobi polynomials.

4.2 Optimization

In this section we will use the same function h as in §4.1.

4.2.1 Caching values of $\gamma_{j,k}$

The following observation was made. In the hp-case, printing input of the function that finds $\gamma_{j,k}^{\triangle}$ shows that there are some redundant calls. In the implementation, a list of $\gamma_{j,k}$ values was already stored (but in a way inefficient for this purpose). See the red graph of Figure 4.2.

4.2.2 Using Remark 2.26

This is fairly self-explanatory. See the green graph of Figure 4.2. After these two optimizations, still 99% of the CPU cycles are dedicated to finding the error $e_r(\triangle)$, with 17% finding $\gamma_{j,k}$ and 82% finding the error given these $\gamma_{j,k}$.

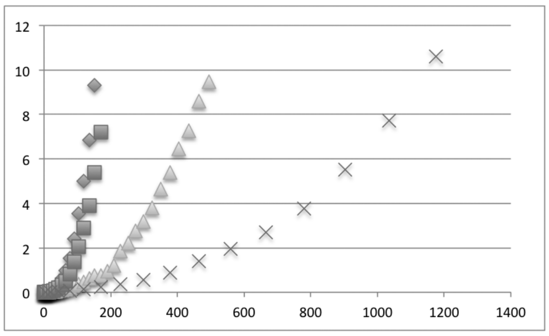
4.2.3 Using Theorem 2.29

In the last few days of this project, it was found that there is a much faster way of finding the error, given $\gamma_{j,k}$. As seen above, this step accounted for 82% of the total computation time and is thus very interesting. After fiddling around, Theorem 2.29 was formulated. This yielded a huge improvement over the previous optimizations. See the purple graph of Figure 4.2.

4.2.4 Parallel computation of $\gamma_{j,k}$

Finding different values of $\gamma_{j,k}$ of the same node $\triangle$ is a task with many independent facets. Therefore, it makes sense to parallelize, yielding a 70% decrease in computation time[2]. See the right of Figure 4.2.

[2]Parallelization was done using the OpenMP[32] compiler directives.



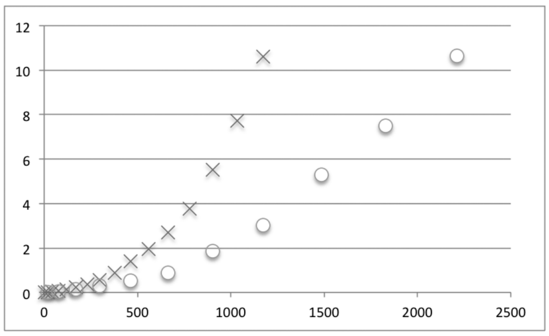


Figure 4.2: Degrees of freedom (x-axis) versus computation time (y-axis) by computing hp-approximants to h. Left: optimizations before parallelization. Right: after. Diamonds: before improvements. Squares: after caching values of $\gamma_{j,k}$. Triangles: after using Remark 2.26. Crosses: after using Theorem 2.29. Circles: after parallel computation of $\gamma_{j,k}$.

4.3 Optimization using a computer cluster

With the optimizations stated, we were able to create a speedup factor between 50 and 500,[3] in the case of an initial partition with one triangle. In case of a polygonal domain, we resort to Algorithm 2.30 (or its counterpart discussed in §5.1). In this setting, we are effectively solving a problem like the one above on each triangle of the initial partition. This is of course another point where parallelization could benefit immensely. Combining both this and the previous concurrent optimization will result in hardly any improvement over choosing just one (assuming the number of triangles in the initial partition is larger than the number of cores available).

The real possibilities of this problem lie in a multiprocessor setting such as a computer cluster: every triangle in the initial partition would then be handled by a separate multicore processor. For problems with large initial partitions, the possbilities are huge. There is very little communication needed between these nodes while processing, as they are solving separate problems. The only communication needed is to divide the problem between nodes at the start, and to communicate the resulting array of leaves at the end.

This is however just a thought; no actual implementation was made.

4.4 Optimization using GPGPU

Numerical integration is a very interesting topic which was hardly discussed (and studied) in the making of this Thesis. This *is* however where 99% of the computation time is spent. Furthermore, numerical integration is a problem that is very open to massive parallelization, as the integration domain can be partitioned in many small elements. On each elements, an integral can be computed. Massively parallel problems like this can be effectively solved using General Purpose GPU programming, or GPGPU. Again, I chose not to pursue this path, as it is too involved.

[3]The lower bound was computed by taking the largest measurement without optimizations (9.3 seconds with 152 DOF) and dividing this by the closest measurement with every optimization (0.142 seconds with 170 DOF) for a factor of 63. The upper bound was found by computing the average speedup factor in each optimization separately, and factoring the results. This yields a factor 463.

5 Discussion

In this Chapter, we will spend some time looking at issues and possible improvements to the two-dimensional algorithms.

5.1 Binev2014 for polygonal domains

The method proposed in §2.4 to use the *hp*-algorithm on each triangle in the initial partition is far from perfect, as each triangle gets an equal amount of iterations and thus refinements. This is of course in contradiction with the notion of adaptive approximation.

In a more practical setting, one may want to iterate until a certain *tolerance* with respect to the total approximation error is reached. The maximum number of iterations – N_{max} – could then be viewed as a way to prevent "infinite" iteration. In light of the above, the algorithm could then be adapted so that given some tolerance ε, the *hp*-algorithm is called on each triangle of the initial partition P with tolerance $\varepsilon/\#P$.

5.2 Validity of *hp*-refining complexity

In [6, p. 16] it is claimed that the complexity of the old version of Algorithm 1.18 is $\mathcal{O}(N^2)$, but only *assuming that the calculation of each local error requires at most constant number of operations*. This is far from true, as the numerically stable way to find this error requires working with the upward recurrence of Theorems 0.33 and 2.15, which is of linear complexity. However, this extra assumption is not explicitly made in the revised version [7]. It is therefore not known whether or not Lemma 1.20 still holds: however, the right graph of Figure 4.1 (on page 40) supports the claim at it *is* in fact of quadratic complexity.

5.3 Conforming partitions in *h*- and *hp*-tree generation

Note: read Appendix A for context. While conforming partitions are interesting in its own right, one may ask why it had to be incorporated in this Thesis. In his original article [4], Binev shows how to apply the principles of Algorithm 1.7 in a typical *h*-refining adaptive Finite Element application. Among a few (unrelated) changes, a modified version of the Algorithm is made in such a way that it creates a conforming partition tree for which the total error is not larger than some target accuracy μ.

The main result of §7 is the following Theorem.

Theorem 5.1. *[4, Thm. 7.2] There are $C_3, c_3 > 0$ such that this modified version of Algorithm 1.7 creates a partition tree T that satisfies*

$$m \leq c_3 \#\texttt{inners}(T) \implies E(T) \leq C_3 E_m,$$

or, in other words, this modified Algorithm is near-optimal in the h-sense.

Of course, one immediately starts to wonder if this property carries over to the more interesting *hp*-case. It is thought[1] that this property does, in fact, not hold for Algorithm 1.18. Further study is needed, but maybe a redefinition of the modified error yields better results.

[1]In other words, not proven.

5.4 This Thesis

The past semester, the writer has devoted a large amount of time to the writing of this Thesis. The initial idea was to understand the theory of near-optimal tree generation in one and two dimensions, *and look at its application in Finite Element Method.* The latter was later adjusted, because the theory of two-dimensional polynomial approximation proved much more involved than anticipated. At first, this seemed like a hindrance but looking back it was a fun and instructive path to take. Moreover, in the continuation of my studies, the Finite Element Method will likely receive much more attention whereas orthogonal polynomials on a triangle will probably not. All in all, I am very grateful for the way this Thesis turned out and all my (adjusted) goals were reached.

One of the things I would like to do in the near future, is to implement the idea of §4.3. There is a computer cluster available (DAS4[28]) on which fellow students have performed computations.

The theory of *hp*-tree generation is not yet fully understood. In particular, preserving near-optimality while creating conforming partitions is a point of further study. Furthermore, the implementation is as of yet not optimal in the sense that it takes quite some time to iterate the *hp*-algorithm. To effectively use it in an applied setting, we will have to spend more time improving.

Part III

Appendices

A Conforming partitions

In multidimensional domain partitions, one has to be more careful. We will explore the notion of *conforming partitions*.

Definition A.1. A *hanging vertex* is a vertex interior to an edge. A partition tree T is said to be *non-conforming* if there is a node $\triangle \in \texttt{leaves}(T)$ with an edge containing a hanging vertex. Thus, a conforming partition tree is one with no hanging vertices.

One often creates non-conforming partitions. The application of our algorithms lie in the adaptive Finite Element Method – we refer to the Applications section on page 3 for an introduction on this subject. The *a priori error bounds* described there are alike to our error functional $e_r(\triangle)$, as they assign an error (bound) to every element in the partition. The constants in this error bound are dependent on (among others) the amount of hanging vertices of a node. If no countermeasures are taken, a single edge can contain an unbounded amount of hanging vertices, and with that, the error bounds tend to infinity. Therefore it is imperative to somehow bound the maximal amount of hanging vertices per edge, the most elegant of course being zero.

Applied to our setting, we are thus looking for a way to find a refinement of some partition which is conforming.

Definition A.2. A *conforming refinement* of a partition tree T is a partition tree $T^c \supset T$ such that T^c is conforming.

Definition A.3. Two nodes $\triangle, \triangle' \in \texttt{leaves}(T)$ are called *neighbours* when $\triangle \cap \triangle'$ is an edge of $\triangle$ or $\triangle'$.

A.0.1 Conforming refinements

Given a conforming partition tree T and a node $\triangle \in \texttt{leaves}(T)$, after subdivision of $\triangle$, one is generally left with a nonconforming tree. Therefore, it is interesting to look at the smallest conforming partition tree in which $\triangle$ has been divided, also known as the *smallest conforming refinement.* We will present an algorithm from [22], which finds exactly this.

Definition A.4. For a triangle $\triangle$ with newest vertex $v(\triangle)$, the edge opposite $v(\triangle)$ is called the *refinement edge* $E(\triangle)$ of $\triangle$.

Algorithm A.5 (Nodal Refinement[22])**.** *Given is a conforming partition tree T on triangular domain D with $\triangle \in \texttt{leaves}(T)$.*

```
refine($T, \triangle$):
  1. If $E(\triangle) \subset \partial D$, subdivide $\triangle$;
  2. Else:
     (a) Let $\triangle'$ be the neighbour of $\triangle$ along $E(\triangle)$;
     (b) If $E(\triangle') = E(\triangle)$, subdivide both $\triangle$ and $\triangle'$;
     (c) Else:
         i. Call refine($T, \triangle'$);
```

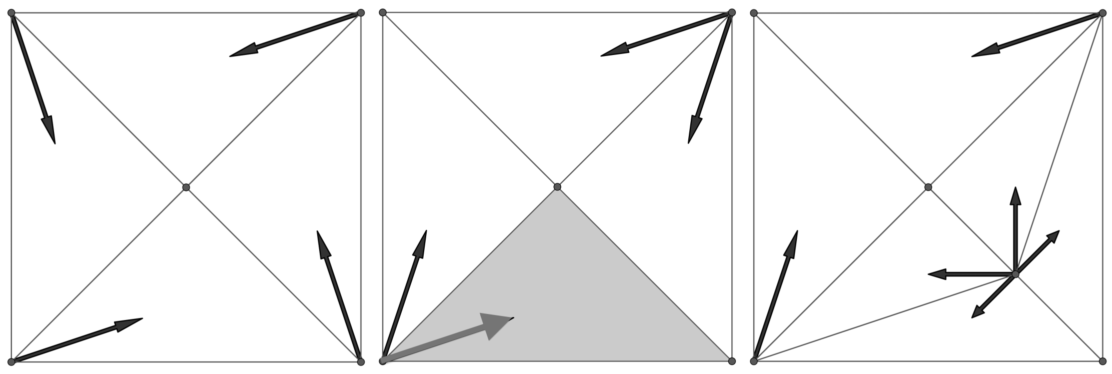


Figure A.1: Left: A counterexample to the unconditional termination of Algorithm A.5 using a polygonal domain and triangulation. Middle: Reordering of newest vertices. Right: Output of Algorithm A.5 on the middle partition applied to the marked node.

> ii. *Let $\triangle''$ be the child of $\triangle'$ with $E(\triangle'') = E(\triangle)$;*
>
> iii. *Subdivide both $\triangle$ and $\triangle''$.*

Theorem A.6 ([22, Thm. 5.1])**.** *Algorithm A.5 terminates and it yields the smallest conforming refinement of T in which $\triangle$ is subdivided.*

In some cases however, we do not have a conforming partition tree. The following algorithm is of help.

Algorithm A.7 (MakeConform[23])**.** *Given a partition tree T, we will construct the smallest conforming refinement.*

```
1. Let $M$ be an empty queue;
2. For each $\triangle \in$ leaves($T$), if $\triangle$ contains a hanging vertex, put $\triangle$ into $M$;
3. While $M \neq \varnothing$:
    (a) Pop $\triangle$ from $M$, and subdivide it;
    (b) For $\hat{\triangle} \in$ children($\triangle$), if $\hat{\triangle}$ contains a hanging vertex, append $\hat{\triangle}$ to $M$;
    (c) For neighbours $\tilde{\triangle}$ of $\triangle$ not in $M$, if $\tilde{\triangle}$ contains a hanging vertex, append it to $M$.
```

Being able to work with triangular domains using conforming partitions is nice, but most real-life applications work with *polygonal* domains.

A.0.2 Matching condition and ensuring conformity

If D is a triangle, Algorithm A.5 is guaranteed to terminate in finitely many steps. However, in a polygonal domain D this is not true in general.

Example A.8. Consider the triangulation depicted in the left of Figure A.1. Calling `refine` on any node invokes an endless recursive loop. A simple solution here would have been to reorder the newest vertices, as seen in the middle and right of Figure A.1.

This gives rise to a question: why did we invoke an endless loop, and why does reordering the vertices solve the problem? A sufficient condition to ensure termination of Algorithm A.5 is the following.

Definition A.9. A partition satisfies the *matching condition* when for every node $\triangle$, the refinement edge is either on ∂D or the neighbour $\triangle'$ along $E(\triangle)$ has refinement edge $E(\triangle') = E(\triangle)$ as well.

Theorem A.10 ([22, Thm. 5.1])**.** *If P is a conforming partition that satisfies the matching condition, then Algorithm A.5 terminates in finitely many steps for any node in the partition.*

In the previous example, the reordering was fairly obvious. In arbitrary partitions, there is guaranteed to be a matching reordering ([5, Lem. 2.1]) but the method referred to is far from simple. The proposed solution is fairly elegant.

Algorithm A.11 (Matching[5, p. 229])**.** *Given an arbitrary conforming partition P, we will construct a conforming refinement which satisfies the matching condition.*

1. *For every triangle in the partition, subdivide it to yield a partition P';*
2. *For each triangle $\triangle_i \in P'$, subdivide it into $\{\triangle_i', \triangle_i''\} := \texttt{children}(\triangle_i)$;*
3. *Set $E(\triangle_i') := \triangle_i' \cap \triangle_i'' =: E(\triangle_i'')$ for each $i \leq \#P'$.*

Theorem A.12. *Algorithm A.11 constructs a conforming refinement which satisfies the matching condition.*

Proof. The resulting partition is a twice-applied uniform refinement of P. It is known [22, Thm. 4.3] that this is a conforming partition.

The matching condition is obviously satisfied by construction. □

As a call to the refinement Algorithm A.5 can have non-local side effects, a triangle marked for subdivision might be subdivided as a side effect. This gives rise to the following algorithm.

Algorithm A.13 (Refine[22])**.** *Let P_0 be a conforming partition that is matching.*

1. *$P := P_0$;*
2. *Iterate until satisfied:*
   (a) *Mark some set of triangles $\overline{M} \subset P$ for subdivision;*
   (b) *For each $\triangle \in \overline{M}$: if $\triangle \in P$, call Algorithm A.5 to receive a conforming refinement, to which P is set.*

Theorem A.14. *[22, Thm. 6.1] If M is the set of triangles on which Algorithm A.5 is called in Algorithm A.13, then for input partition P_0 and output partition P, it holds that*

$$\exists C \quad \#(P \setminus (P \cap P_0)) \leq C\#M,$$

where C is a constant dependent on P_0 only. In other words, given P_0, the total amount of subdivisions made is bounded by a constant times the intended amount of subdivisions.

B Condition number of the 2D monomial basis

The claim made in Theorem 2.13 pertains to the mass matrix (cf. the notational remark 2.10)

$$H_{i,j} := \langle \varphi_i, \varphi_j \rangle = \iint\limits_T x^{k+m} y^{l+n} dy dx, \quad \varphi_i = \varphi_{k,l} = x^k y^l, \quad \varphi_j = \varphi_{m,n} = x^m y^n$$

where T is some arbitrary triangle. Lemma 2.12 tells us that this T is free to be chosen, so take T the triangle spanned by vertices $(0,0), (1,0), (0,1)$.

Then integration by parts yields

$$H_{i,j} = \frac{(k+m)!(l+n)!}{(k+l+m+n+2)!}.$$

Let n be fixed as the highest degree of basis elements. By Lemma 2.9, we know that this basis has $(n+2)(n+1)/2$ elements in an $n+1$ by $n+1$ "triangular form":

$$\begin{array}{cccc} x^3 & & & \\ x^2 & x^2y & & \\ x & xy & xy^2 & \\ 1 & y & y^2 & y^3 \end{array}$$

We need to order this basis in such a way that basis elements can be uniquely identified by a single index. For this, we use the Cantor pairing function from §0.2.2.

With the Cantor pairing function, we can form a mass matrix H by:

$$(a,b) = \pi^{-1}(i), \quad (c,d) = \pi^{-1}(j), \quad H_{i,j} = \langle x^a y^b, x^c y^d \rangle = \frac{(a+c)!(b+d)!}{(a+b+c+d+2)!}.$$

Finding the condition number of this dense matrix is very hard analytically. Therefore the author chose to work out the first few degrees using Python and `numpy.linalg.cond`. The conclusion is Figure B.1. One can easily see that, at least for $n \leq 9$, the condition number of $\mathcal{M}_n$ grows exponentially.

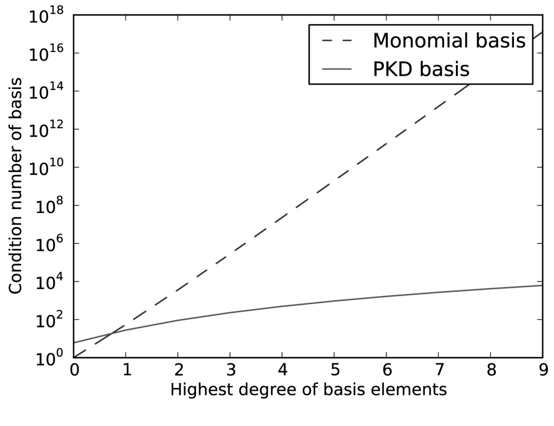


Figure B.1: Condition number of the bases presented in Lemma 2.9 and Theorem 2.19.

C On Algorithm 1.6

Let T_j be the tree created by Algorithm 1.6 at step j. We will prove that for every $0 < C_1 \le 1 \le C_2$, there is a function $f \in L^2(0,1)$ with

$$\exists j \in \mathbb{N}, \exists T_* : \#T_* \le C_1 j, \quad E(T_j) > C_2 E(T_*).$$

In other words, we will show that Algorithm 1.6 is not near-optimal. To this end, we will be constructing a function $f = f(n,m)$ and approximate it with piecewise constant polynomials, in such a way that the Greedy approach will never reduce the total error $E(T_j) = \|f\|_2$. Contrastingly, we'll construct a tree T_* (i.e., a dyadic partition of $[0,1]$) with small error compared to $E(T_j)$.

Choose $n, m \in \mathbb{N}$ such that

$$\frac{2C_2}{n+2} < 1, \quad 2n \le C_1(n+m).$$

Furthermore, let $g \in L^2(0,1)$ with

$$g(x) := \begin{cases} 1 & x \in [0, \frac{1}{2}], \\ -1 & x \in [\frac{1}{2}, 1]. \end{cases}$$

Then $\|g\|_2 = 1$ and its polynomial of best approximation is the zero polynomial. Lastly, let $h \in L^2(0,1)$ be

$$h(x) := g(2^{m-1}x \mod 1).$$

Then $\|h\|_2 = 1$, and for any element $\triangle$ in any dyadic refinement of $[0,1]$ with m elements, the constant polynomial of best approximation on $\triangle$ equals the zero polynomial.

Define

$$f(x) := \begin{cases} 2^{i+1}\sqrt{i+1}\, g(2^{i+1}x - 1) & x \in [2^{1-i}, 2^{-i}], \quad i \in \{0, \dots, n-1\} \\ 2^n \sqrt{n+1}\, h(2^n x) & x \in [0, 2^{-n}]. \end{cases}$$

Then for all the above intervals, the constant polynomial of best approximation is the zero polynomial. Furthermore, for $i \in \{0, \dots, n-1\}$, $\|f\|^2_{2,[2^{1-i},2^{-i}]} = i+1$ and $\|f\|^2_{2,[0,2^{-n}]} = n+1$ so that $\|f\|^2_{2,[0,1]} = \sum_{i=0}^n i+1 = (n+1)(n+2)/2$.

As the local error of approximation is increasing in i, for $j \le n$, the Greedy algorithm will always subdivide the leftmost interval, yielding $[0, 2^{-n}]$ as the leftmost element of the partition after n iterations. The next m subdivisions will also yield no error reduction (as per construction of h), and thus, for $j \le n+m$, the total error $E(T_j)$ equals $\|f\|_{2,[0,1]}$.

Consider the partition tree T_* found by the following: for $i \in \{0, \dots, n-1\}$, subdivide $[0, 2^{-i}]$ and subsequently $[2^{-(i+1)}, 2^{-i}]$. After $2n$ subdivisions, the local error of approximation is reduced to 0 for all elements but $[0, 2^{-n}]$. Hence,

$$E(T_*) = \|f\|^2_{2,[0,2^{-n}]} = n+1 = \frac{n+1}{(n+1)(n+2)/2}\|f\|^2_{2,[0,1]} = \frac{2}{n+2}\|f\|^2_{2,[0,1]}.$$

By virtue of the choices of n and m, we conclude that Algorithm 1.6 is *not* near-optimal.

D The 2D implementation

All except the first module come in the form of a header and a source file.

`types` provides the used types and structures of this project;

`workspace` provides a data type containing the polygonal roots, partition trees, an array of leaves and the matrix of edges between triangles;

`triangulate` implements the Ear-Clipping triangulation algorithm;

`partition` sets up and manipulates partition trees;

`pkd` provides evaluation of the PKD-polynomial basis;

`edge` provides functionality to manipulate the edge matrix;

`pair` provides us with pairing functions from §0.2. This module contains no interesting code whatsoever and will not be further discussed;

`tri` contains functionality to transform between triangles;

`error` contains functionality pertaining to finding nodal error of approximation.

`types`

The main difference between this implementation and its previous ones is the addition of a workspace structure. This was done – partly inspired by the GNU Scientific Library – to contain all information in a single structure.

Nomenclature of the h- and hp-tree struct members loosely follow that of the thesis (for instance, `tehp` stands for tilde error hp, or $\tilde{e}^{hp}$). The arrays `coeffs` are stored to be able to plot the resulting approximation.

Note that the Newest Vertex Bisection, and the notion of *newest vertex* and *refinement edge*, are implicitly incorporated in this implementation, as the first point in a triangle denotes its newest vertex, and the second and third span its refinement edge.

`workspace`

The `workspace` struct is set up with a (dynamically reallocating) array of points, array of triangles, array of h- or hp-trees (where every element of this array is one element of the initial triangulation), and array of leaves for quick traversing.

This module did not prove to be particularly challenging.

`triangulate`

Aside from the fact that triangulation by the ear-clipping method (see Algorithm 2.4) is very interesting, this module contains no surprises. The `sameside` method was found in [33] and is perhaps the most interesting to note.

`partition`

This module implements Algorithms A.7, A.5 and A.13. This is also the module which relies on finding neighbours along a common edge in $\mathcal{O}(1)$ time (by means of the edge matrix). Of these algorithms, Algorithm A.7 proved the most difficult. In particular, finding out if a triangle contains a hanging node in a time-efficient manner proved to be hard (this problem was later solved after finding [23, Prop. 3.1]).

`pkd`

Implementing this module (and in particular, the method `pkd_eval_square()`) proved to be exceedingly hard, as sources surrounding this subject were very scarce. Also, profiling using the `valgrind` [34] and `kcachegrind` [27] tools shows that this is – just as in the one-dimensional case – the module where most CPU time is spent (and thus being the biggest candidate for optimization).

`edge`

The edge matrix is a square matrix with `npoints` columns. When a point is added to the partition as a result of a subdivision, both a row and a column is added to this matrix. Representing the edge matrix by a two-dimensional array internally means adding an element to every row, and creating one new row. This is of $\mathcal{O}(\texttt{npoints})$ complexity. We can achieve $\mathcal{O}(1)$ complexity by storing this matrix as a one-dimensional array with appropriate ordering (see Szudzik pairing function, §0.2.1).

`tri`

While in one dimension, 'volume' of a node $[a, b]$ is just $b - a$ and is easily calculated. In two dimensions however, the volume of a node is slightly more intricate (see Definition 2.21) and was stored in the struct of the node to make a tiny optimization.

`error`

This module finds the local error of approximation of Definition 1.13 and is immensely more involving than in the one-dimensional case. Of every module in the project, finding the error by far cost most time to implement correctly.

Integration over triangles, both arbitrary and predetermined, is in practice harder than integrating over a known square. We will therefore transform the requested integrals (2.2) and (2.3) to a square integral to ease implementation.

Consider

$$\gamma_{j,k} := \frac{(2j+1)(j+k+1)}{\operatorname{vol}(\triangle)} \iint\limits_{\triangle} f(x,y) Q^{\triangle}_{j,k}(x,y)\,dydx,$$

with

$$\iint\limits_{\triangle} f(x,y) Q^{\triangle}_{j,k}(x,y)\,dydx = 2\operatorname{vol}(\triangle) \iint\limits_{\hat{\triangle}} f(G(x,y)) Q_{j,k}(x,y)\,dydx.$$

Transforming to the square using Remark 2.17 yields

$$\iint_{\hat{\triangle}} f(G(x,y))Q_{j,k}(x,y)dydx = \iint_{\square} (1-t)/8(f\circ G\circ F)(s,t)Q^{\square}_{j,k}(s,t)dtds.$$

Hence, combining, we find

$$\gamma_{j,k} = (2j+1)(j+k+1)/4\iint_{\square}(1-t)(f\circ G\circ F)(s,t)Q^{\square}_{j,k}(s,t)dtds. \tag{D.1}$$

Futhermore,

$$\iint_{\triangle}[f(x,y)-p_{n(r)}(x,y)]^2dydx = 2\,\mathrm{vol}(\triangle)\iint_{\hat{\triangle}}\left[(f\circ G)(x,y)-\sum_{j=0}^{n(r)}\sum_{k=0}^{n(r)-j}\gamma_{j,k}Q^{\hat{\triangle}}_{j,k}(x,y)\right]^2 dydx \tag{D.2}$$

$$= \mathrm{vol}(\triangle)/4\iint_{\square}(1-t)\left[(f\circ G\circ F)(s,t)-\sum_{j=0}^{n(r)}\sum_{k=0}^{n(r)-j}\gamma_{j,k}Q^{\square}_{j,k}(s,t)\right]^2 dtds.$$

Throughout the implementation, we used GSL's `gsl_integration_gags()` method. This is a method that adaptively partitions the domain to find the integral. We quote [31]:

> This function applies the Gauss-Kronrod 21-point integration rule adaptively until an estimate of the integral of f over (a,b) is achieved within the desired absolute and relative error limits, `epsabs` and `epsrel`. The results are extrapolated using the Wynn epsilon-algorithm, which accelerates the convergence of the integral in the presence of discontinuities and integrable singularities. The function returns the final approximation from the extrapolation, `result`, and an estimate of the absolute error, `abserr`. The subintervals and their results are stored in the memory provided by `workspace`. The maximum number of subintervals is given by `limit`, which may not exceed the allocated size of the workspace.

To find this integral, we used a `limit` value of 1000, the default value. Furthermore, `epsrel` was 0 (so that only absolute errors are considered) with `epsabs` equal to 10^{-15}, which seemed to work well for most purposes. It was only in isolated cases that this proved insufficient.

Usage

We assume a function

```
double f( double x, double y);
```

to be defined in `main.h`. We can then initialize the workspace using

```
workspace *w = workspace_init();
```

and then define the polygon by making a `point` array (with elements in counter-clockwise order)

```
point points[4] = { {0, 0}, {1, 0}, {1, 1}, {0, 1} };
w->npoly = sizeof( points)/sizeof( points[0]);
for( i = 0; i < w->npoly; i++)
  workspace_add_point( w, points[i]);
```

We can then choose to triangulate the polygon by

```
triangulate( w);
```

or manually define a triangulation by, for instance:

```
tri *tris[2] = { tri_create( w, 0, 1, 2), tri_create( w, 2, 3, 1)};
for( i = 0; i < 2; i++)
  workspace_add_tri( w, tris[i]);
```

We then setup the partition (i.e., further initialize the workspace based on the current triangulation) by

```
partition_setup( w, IS_HP);
```

where `IS_HP` is of course 0 when we want to approximate using h-refinement and 1 otherwise. After this, we can choose to create a matching partition – of course, this is only needed if the original partition was not already matching:

```
partition_match( w);
```

Then we can call the tree-generating algorithm by

```
if( IS_HP)
  treegen_hp( w, STEPS);
else
  treegen_h( w, STEPS, DOF);
```

where `DOF` stands for *degrees of freedom* and `STEPS` is the number of iterations to use.

After this, we can choose to make a conforming partition by

```
partition_make_conform( w);
```

In any case, one might want to plot the resulting approximation. To accomplish this, we print by

```
workspace_print_plot( w);
```

and free everything by

```
workspace_free( w);
```

To plot the approximation, one calls

```
./main | python plottri.py
```

which pipes the output of `workspace_print_plot( w)` to the plotter script.

Bibliography

[1] R.M. Aarts, C. Bond, P. Mendelsohn, E.W. Weisstein, *Remez Algorithm*, MathWorld, `http://mathworld.wolfram.com/RemezAlgorithm.html`.

[2] M. Abramowitz, I.A. Stegun, *Handbook of Mathematical Functions*, National Bureau of Standards Applied Mathematics Series 55, 1972.

[3] G.B. Arfken, H.J. Weber, *Mathematical Methods for Physicists*, Elsevier Academic Press, 2005.

[4] P. Binev, R. DeVore, *Fast computation in adaptive tree approximation*, Springer-Verlag, 2004.

[5] P. Binev, R. DeVore, Wolfgang Dahmen, *Adaptive Finite Element Methods with convergence rates*, Numer. Math. (2004), 2004.

[6] P. Binev, *Instance optimality for hp-type approximation*, Oberwolfach Report 39, 2013.

[7] P. Binev, *Tree approximation for hp-Adaptivity*, private correspondence, April 2014.

[8] P. Binev, *Adaptive Methods and Near-Best Tree Approximation*, Oberwolfach Report 29, 2007.

[9] R. Bürger, M. Sepúlveda, T. Voitovich, *On the PKD Hierarchical Orthogonal Polynomial Bases for the DG-FEM*, 2009.

[10] M.D. Choi, *Tricks or Treats with the Hilbert Matrix*, Amer. Math. Monthly 90, 301-312, 1983.

[11] B. Delaunay, *Sur la sphère vide. A la mémoire de Georges Voronoï*, Bulletin de l'Académie des Sciences de l'URSS, Classe des sciences mathématiques et naturelles, 1934.

[12] W. Dörfler, V. Heuveline, *Convergence of an adaptive hp finite element strategy in one space dimension*, Applied Numerical Mathematics 57 (2007), 2006.

[13] M. Dubiner, *Spectral Methods on Triangles and Other Domains*, Plenum Publishing, 1991.

[14] D. Eberly, *Triangulation by Ear Clipping*, `http://www.geometrictools.com/Documentation/TriangulationByEarClipping.pdf`.

[15] T. Koornwinder, *Two-variable analogues of the classical orthogonal polynomials*, Theory and Applications of Special Functions, 1975.

[16] J.M. Melenk, B.I. Wohlmuth, *On residual-based a posteriori error estimation in hp-FEM*, Advances in Computational Mathematics, 15: 311-331, 2001.

[17] W.F. Mitchell, *A Comparison of Adaptive Reminement Techniques for Elliptic Problems*, ACM Trans. Math. Software 15(4), 1989.

[18] J. Proriol, *Sur une familie de polynomes á deux variables orthogonaux dans un triangle*, C. R. Acad. Sci. Paris 257, 1957.

[19] B. Rynne, M.A. Youngson, *Linear Functional Analysis*, Springer, 2008.

[20] C. Schwab, *p- and hp-finite element methods*, Oxford University Press, 1998.

[21] E.G. Sewell, *Automatic Generation of Triangulations for Piecewise Polynomial Approximation*, Thesis, Purdue University, 1972.

[22] R.P. Stevenson, *The completion of locally refined partitions created by bisection*, Math. Comp. 77 (2008), 2007.

[23] R.P. Stevenson, *Optimality of a Standard Adaptive Finite Element Method*, Found. Comput. Math. 245-269, 2007.

[24] E. Süli, D. Mayers, *An Introduction to Numerical Analysis*, Cambridge University Press, 2003.

[25] M. Szudzik, *An Elegant Pairing Function*, `http://szudzik.com/ElegantPairing.pdf`.

[26] G. van Winckel, *Orthogonal polynomials on the triangle*, `http://www.scientificpython.net/pyblog/orthogonal-polynomials-on-the-triangle`.

[27] Josef Weidendorfer, *kcachegrind*, `http://kcachegrind.sourceforge.net/html/Home.html`.

[28] *DAS4 cluster website*, `http://www.cs.vu.nl/das4/`, Vrije Universiteit Amsterdam, 2012.

[29] *Inverse Hilbert function*, `http://docs.scipy.org/doc/scipy/reference/generated/scipy.linalg.invhilbert.html`, Scipy Documentation, 2013.

[30] *Legendre Polynomials*, `https://www.gnu.org/software/gsl/manual/html_node/Legendre-Polynomials.html`, GSL Documentation.

[31] *Numerical Integration*, `https://www.gnu.org/software/gsl/manual/html_node/Numerical-Integration.html`, GSL Documentation.

[32] *OpenMP*, `http://openmp.org/wp/`.

[33] *Point in triangle test*, `http://www.blackpawn.com/texts/pointinpoly/default.html`.

[34] *Valgrind*, `http://valgrind.org/`.

Populaire samenvatting

Adaptieve benadering is een onderzoeksveld dat zich bezig houdt met het benaderen van functies, op zo'n manier dat de methode *adaptief* wordt aangepast aan het probleem. We kijken dus op sommige delen door een 'hogere resolutie camera' dan op andere. Adaptieve benaderingsmethodes laten zich veelal beschrijven door een *boom* die de adaptieve keuzes bijhoudt. Er bestaan veel van dit soort bomen en vaak is men gemoeid met het vinden van de *optimale* boom. Omdat dit voor problemen met relatief hoge resolutie al snel 'ondoenlijk' wordt, is er de afgelopen jaren onderzoek gedaan naar het vinden van bijna-optimale bomen die 'snel' gevonden kunnen worden. De benaderingsfout die hoort bij zo'n bijna-optimale boom is 'bijna even goed' als de optimale boom met iets minder hoge resolutie.

In dit verslag bekijken we twee methodes – eentje uit 2004, de ander is zelfs nog in ontwikkeling – voor het vinden van bijna-optimale bomen. We hebben een functie in 1 of 2 variabelen, gedefinieerd op haar functiedomein: een interval in 1 dimensie, een polygon[1] in twee. Vervolgens pakken we een *initiële partitie* – opdeling van het domein in intervallen of driehoeken – en blijven deze adaptief *verfijnen* door herhaald een element uit de partitie (opdeling van je domein) te kiezen en deze doormidden te delen[2]. Hierbij is het *kiezen* nog helemaal niet zo makkelijk: als je steeds het element kiest waarop de benadering het slechtste is (de *Greedy approach*), dan hoeft de uiteindelijke boom *niet* optimaal te zijn!

Maar wat is nou eigenlijk die benadering? In dit verslag beperken we ons tot *polynomiale* benaderingen: je zoekt, gegeven een bepaalde graad p, het best-benaderend polynoom van graad p over een element uit de partitie. In de eerste methode die we bekijken staat deze graad vast (je maakt bijvoorbeeld een stuksgewijs lineare benadering van de beginfunctie). Omdat de grootte van de elementen uit de partitie – h – verschillend is, wordt dit ook wel h-type boomgeneratie genoemd. De tweede methode staat een variabele graad per element uit de partitie toe en heet daarom ook wel hp-type boomgeneratie. Het begrip polynomiale graad wordt iets gegeneraliseerd tot *vrijheidsgraden*, of in andere woorden, het aantal variabelen dat 'vrij' te kiezen is.

Omdat dit project de climax van zowel Wiskunde als Informatica is, is er ook veel tijd besteed aan het belichten van de meer praktische kant van het verhaal. Naast het verslag is er ook een volledig werkende implementatie van alle theorie die beschreven wordt. Voor de lezer betekent dit dat er gemakkelijk mooie voorbeelden gevisualiseerd kunnen worden. We bekijken de functie

$$f : [0,5] \to \mathbb{R} : x \mapsto \begin{cases} e^x/e^3 & x < 3 \\ \sin(\pi x/2) & x \geq 3 \end{cases}.$$

en gaan deze benaderen met stuksgewijs lineaire polynomen (dus twee vrijheidsgraden per element uit de partitie), gebruikmakend van de zojuist beschreven h-type methode. Zie Figure D.1. Omdat de functie discontinu is in het punt $x = 3$, is de benadering rond dit punt erg slecht. Je ziet dan ook dat de partitie vooral híer verfijnd wordt. Als we een hp-benadering maken (en dus op elk stukje een verschillende polynomiale graad toestaan), ziet het er uit zoals in Figure D.2. Weer is er veel aan de hand rond het punt $x = 3$. In §3.1 op pagina 31 van dit verslag gaan we

[1]Een polygon is een object dat wordt gekarakteriseerd door een eindig aantal punten die verbonden zijn door lijnstukken, zó dat het geheel gesloten is.

[2]Hoewel dit in 1 dimensie het opdelen van zo'n interval eigenlijk op 1 manier kan, zijn er in twee dimensies natuurlijk meedere manieren om een driehoek doormidden te delen. De methode die eigenlijk iedereen gebruikt is de Newest Vertex Bisection, maar voert helaas te ver voor deze korte introductie.

in meer detail in op dit probleem.

Tot slot bekijken we een *hp*-benadering van een tweedimensionale functie. Beschouw

$$g(x, y) := \begin{cases} 1 & x + y > 1.1 \\ 0 & x + y \leq 1.1 \end{cases},$$

waarbij het domein in het middelste plaatje van Figure D.3 te zien is. We gaan deze functie benaderen door stuksgewijs polynomen, dus over elke driehoek komt zo'n polynoom te liggen. De eerste benadering – eentje met een constant polynoom over elke driehoek – is te zien in de linkerkolom van Figure D.4. Als we daarna een aantal driehoeken opdelen (óf de polynomiale graad op een driehoek verhogen!), krijg je benaderingen zoals midden en rechts te zien is.

In de implementatie van alle theorie is ook aandacht besteed aan optimalisatie: je wil niet alleen dat het werkt, maar ook dat de gebruiker niet te lang hoeft te wachten. Een van de aandachtspunten was dan ook *parallellisatie*: gebruik maken van het feit dat computers heden ten dage veelal meerdere dingen *tegelijk* uit kunnen rekenen (op een zogeheten *multi-core* processor).

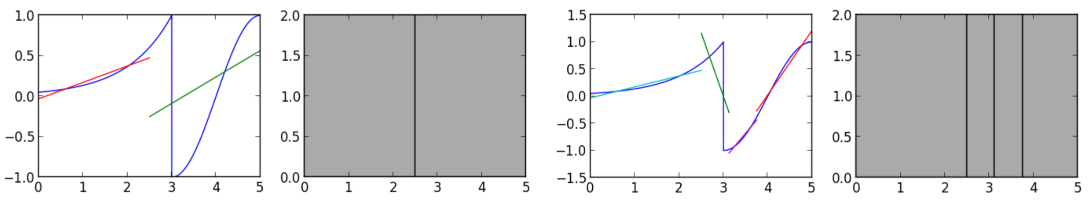


Figure D.1: Een stuksgewijs lineaire benadering van f, geconstrueerd met de methode van h-type boomgeneratie. We zien de eerste twee stappen van het algoritme, overeenkomend met een totaal van 4 (links) en 8 (rechts) *vrijheidsgraden* – het totaal aantal vrij te kiezen variabelen. De grijze plot geeft de distributie van vrijheidsgraden weer.

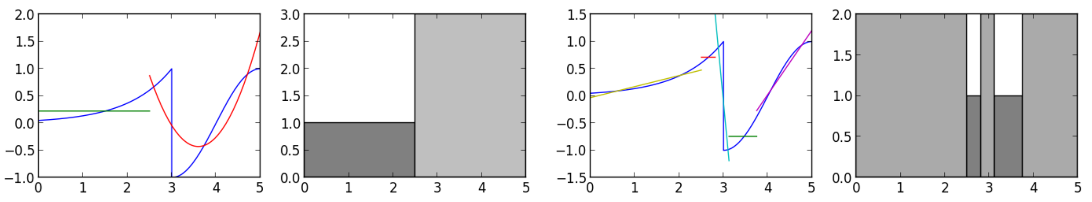


Figure D.2: Een benadering van f door op ieder element uit de partitie (opdeling van domein) een verschillende polynomiale graad toe te staan. De totale hoeveelheid vrijheidsgraden is even veel als in Figuur D.1.

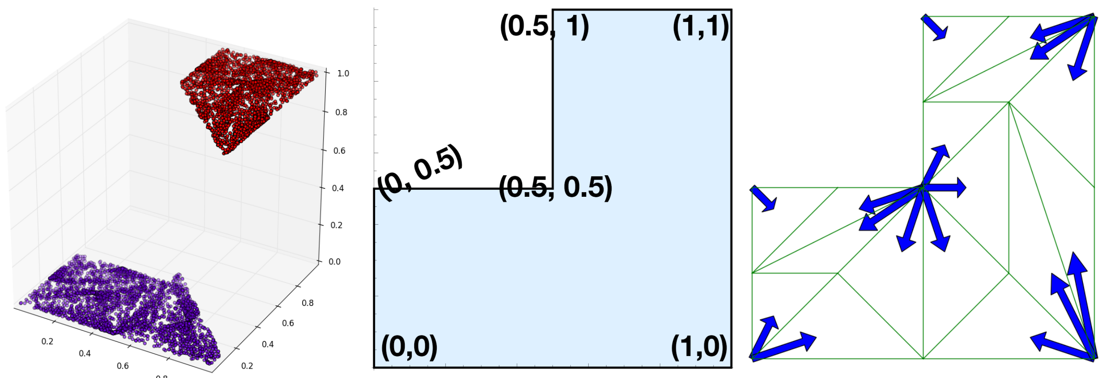


Figure D.3: V.l.n.r: een plot van de functie g; haar functiedomein; de *initiële partitie* (eerste opdeling van het domein). De blauwe pijlen hebben te maken met de methode om een driehoek doormidden te delen (de Newest Vertex Bisection).

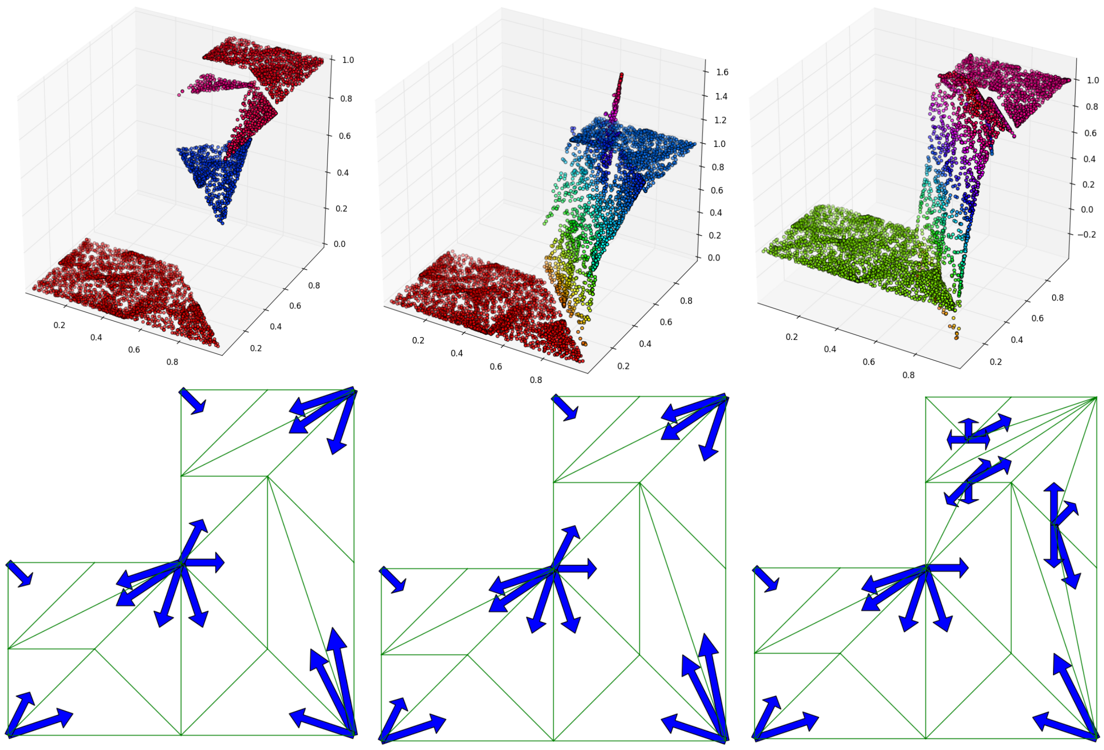


Figure D.4: Bovenste rij: plots van de benadering van g na herhaald toepassen van de *hp*-methode. Onderste rij: de partitie (opdeling) waarover de polynomen liggen.