Generalization of Whitney Decomposition in Three Dimensions and Its Implementation in Mathematica

Advisor: Antti Rasila
antti.rasila@gtiit.edu.cn

Student: Yu Miao
miao07685@gtiit.edu.cn

Guangdong Technion
Mathematics with Computer Science

July 8, 2023

Abstract

The Whitney decomposition of a domain in R^n is an important tool in real and harmonic analysis. This research proposal aims to explore the generalization of Whitney decomposition in two and three dimensions, starting with the two-dimensional case. The project will develop a Mathematica implementation of the decomposition, drawing inspiration from the implementation of Whitney decomposition in the plane in a "Zipper" numerical conformal mapping package by Donald E. Marshall. Under the guidance of Associate Professor Antti Rasila at GTIIT, student Yu Miao will participate in this research project

Contents

1 Introduction

[1] The Whitney decomposition has been widely used in real and harmonic analysis for studying the relative geometry of planar domains. In this research, we will extend the existing results on two-dimensional domains to three-dimensional domains by considering cubes in place of squares. This generalization will provide new insights into the geometry of three-dimensional domains and their properties.

1.1 Background and Motivation

Whitney decomposition enables the representation of a bounded planar domain as a union of non-overlapping closed squares. The decomposition can be naturally extended to higher dimensions, with cubes replacing squares.

In the two-dimensional case, we have a series of closed squares Q_j^k, $k \in Z$, $0 \leq j \leq N_k$ referred to as Whitney squares. These squares exhibit certain properties: each square has a side length of 2^{-k}, $k \in Z$, the union of all squares equates to the domain, and the distance from each square to the boundary of the domain is less than four times the side length of the square.

Whitney squares are understood as approximations for quasihyperbolic balls, each square containing and being contained in a quasihyperbolic ball of different radii. The union of squares closely approximates the shape of the boundary of the domain. This brings us to an important concept: the integer sequence N_k provides significant information about the domain's geometry.

If certain geometric conditions hold, this sequence gives us insight into the Hausdorff dimension of the boundary of the domain. This can even be reversed under some circumstances: if the boundary of the domain is a compact set of zero measure, we can get an upper bound for N_k in terms of the Hausdorff dimension.

Whitney decomposition also serves to calculate upper bounds for quasihyperbolic distances between two points in a domain. To do this, we pick a collection of Whitney squares such that there exists a curve within the union of these squares that includes both points. The cardinality of this collection of squares gives us a comparable upper bound for the quasihyperbolic distance between the points.

Motivation for this research lies in extending this two-dimensional approach to three dimensions. If successful, we could reduce of computational complexity when approximating areas in two dimensions or volumes in three dimensions. Implementing this algorithm in Mathematica will facilitate easier analysis of complex geometric shapes.

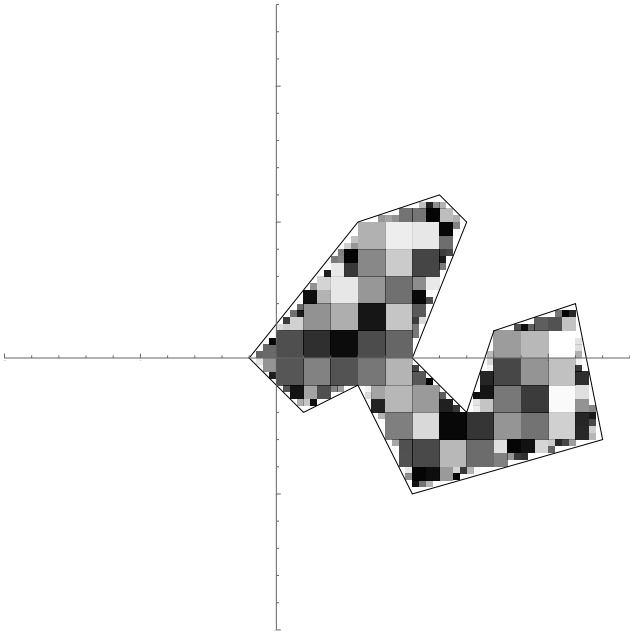
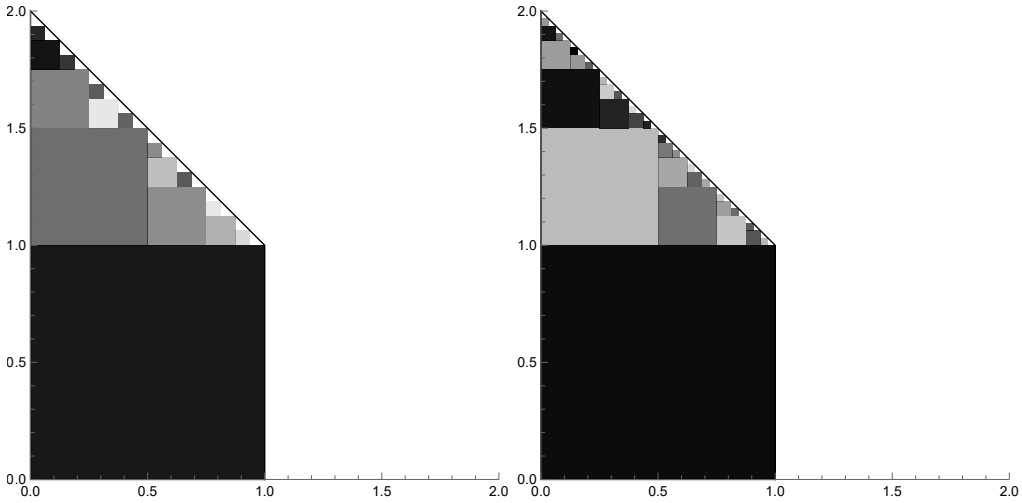


Figure 1.1.1: a replica to the example given in the book [1]



(a) minimal side length of 2^{-4} (b) minimal side length of 2^{-5}

Figure 1.1.2: Examples of Whitney decomposition in 2D

1.2 Research Questions and Objectives

- To create an implementation of the Whitney decomposition algorithm in Mathematica, inspired by Donald E. Marshall's Fortran code for the two-dimensional case.
- To generalize the concept of Whitney decomposition from two dimensions to three dimensions, exploring potential alterations required for such a transformation.

2 Auxiliary Implementations

2.1 PNPOLY - Point Inclusion in Polygon Test by W. Randolph Franklin

[2]

In our exploration of 2D implementation of the Whitney decomposition, we encounter the PNPOLY algorithm by W. Randolph Franklin, which is a robust and efficient method to determine if a given point lies inside a polygon. This algorithm has been widely used in Geographic Information Science (GIS) for spatial locality queries and possesses significant utility across a multitude of computational and real-world applications.

The PNPOLY algorithm works by casting a semi-infinite ray horizontally from the point in question and counting the number of polygon edges that the ray intersects. If the count is odd, the point lies inside the polygon. If it's even, the point lies outside. This method, despite its simplicity, manages to account for various cases, such as when the point lies on the edge or vertex of the polygon.

Furthermore, our implementation provides a method for determining if a point $x = (x_1, x_2)$ lies on the boundary of a polygon P, identified by the points $p(j)$. This is achieved by examining each pair of points, $p(j)$ and $p(j+1)$, in the polygon and verifying the following equation:

$$|p(j) - x| + |x - p(j+1)| = |p(j) - p(j+1)|$$

If the equation holds true (within a reasonable tolerance, e.g., 10^{-6}), the point x is situated on the line segment connecting $p(j)$ and $p(j+1)$. Otherwise, the point x does not reside on the boundary.

2.2 Numerical Conformal Mappings Software by Donald E. Marshall

[3]

In exploring the implementation of Whitney decomposition on the complex plane, I was inspired by the algorithm implemented in FORTRAN by Donald E. Marshall.

The code begins by reading in files containing the boundary points of a region and the Whitney squares. It then establishes the maximum and minimum x and y values that will ultimately define the complex grid for the decomposition. After initializing the center of the grid and its size, the program enters a nested loop structure where it iteratively refines the decomposition level by level.

At each level, it examines the candidate squares (newly divided smaller squares from the previous level) to decide whether they should be included in the next level of decomposition. This decision is made by a series of logical checks concerning the relative position of the squares to the region boundary and their interaction with boundary points.

It also employs the winding number concept to determine if a point is inside a polygon. This is done by calculating the net number of times the path winds around the point.

3 Methodology

3.1 2D Generalization of Whitney Decomposition

Our goal is approximating a polygon using the largest possible squares of a side length 2^{-k}, $k \in Z$.

Let's define a polygon D1, for instance, $D1 = \{\{1, 0\}, \{1, 1\}, \{0, 2\}, \{0, 0\}, \{1, 0\}\}$.

We have a given function PNPoly1 that takes a point and a polygon as inputs, and returns true if the point is inside the polygon, otherwise returns false.

Create a list called listCandidates, and store all squares of side length 1 within a certain range, for instance, in this case, of [-5, 5] x [-5, 5] on the 2D plane. Create a list called listOutput.

Use PNPoly1 to check whether all four vertices of each square in listCandidates are inside the polygon D1. If all vertices are inside D1, add the square to listOutput and remove it from listCandidates.

Replace each square in listCandidates with four smaller squares of side length 1/2.

Use PNPoly1 again to check whether all four vertices of each smaller square in listCandidates are inside the polygon D1. If all vertices are inside D1, add the smaller square to listOutput.

Repeat the process: In each iteration, replace the remaining squares in listCandidates with smaller squares of side length equal to half of the original side length.

Here is the Mathematica implementation of this algorithm.

Define a polygon by its vertices.

```
D1 = {{1, 0}, {1, 1}, {0, 2}, {0, 0}, {1, 0}};
```

This function checks if a point is inside a polygon using the PNPoly algorithm by W.Randolph Franklin.

```
(*Is a point (x,y) inside the polygon PP?*)
D1={{1,0},{1,1},{0,2},{0,0},{1,0}};
PNPoly1[{x0_,y0_},PP_]:=Module[{mm,nn,j,c=False,onBoundary=False,epsilon
    =10^-6},{nn,mm}=Dimensions[PP];
nn=nn-1; (*Subtracting 1 due to the same first and last point*)

(*Test for point inside the polygon*)
For[j=1,j<nn,If[(PP[[j+1,2]]>y0)!=(PP[[j,2]]>y0)&&(x0<(PP[[j,1]]-PP[[j
    +1,1]]) (y0-PP[[j+1,2]])/(PP[[j,2]]-PP[[j+1,2]])+PP[[j+1,1]]),c=!c];
j++];
(*Test for point on the boundary*)
For[j=1,j<nn,nextIndex=Mod[j,nn-1]+1;
distA=Norm[{x0,y0}-PP[[j]]];
distB=Norm[{x0,y0}-PP[[nextIndex]]];
distC=Norm[PP[[j]]-PP[[nextIndex]]];
If[Abs[distA+distB-distC]<=epsilon,onBoundary=True];
j++];
Or[c,onBoundary]];
```

This function checks if all vertices of a square are inside the polygon using PNPoly1. It takes in the vertices of a square and a polygon as inputs and returns true if all vertices are inside the polygon, otherwise returns false.

```
testSquare[vertices_, polygon_] := Module[{output1},
  output1 = PNPoly1[#, polygon] & /@ vertices;
  And @@ output1
  ]
```

This function splits a square into four smaller squares. It takes in a square and the side length of the square as inputs and returns four smaller squares.

```
splitSquare[square_, len_] :=
 Module[{lowerLeft, upperLeft, lowerRight, upperRight, half},
  half = len/2;
  lowerLeft = square[[1]];
  upperLeft = square[[3]];
  lowerRight = square[[2]];
  upperRight = square[[4]];
  {{lowerLeft, lowerLeft + {half, 0}, lowerLeft + {0, half},
    lowerLeft + {half, half}}, {lowerRight + {-half, 0}, lowerRight,
    lowerRight + {-half, half}, lowerRight + {0, half}},
```

```
   {upperLeft + {0, -half}, upperLeft + {half, -half}, upperLeft,
    upperLeft + {half, 0}}, {upperRight + {-half, -half},
    upperRight + {0, -half}, upperRight + {-half, 0}, upperRight}}]
```

This function applies the splitSquare function to all squares in a list. It takes in a list of squares and the side length as inputs and returns a list of the resulting smaller squares.

```
splitAllSquares[listOfSquares_, len_] := Module[{
   length,
   output1
   },
  length = Length[listOfSquares];
  output1 = Table[
    F = splitSquare[#, len] &;
    F[listOfSquares[[j]]],
    {j, 1, length, 1}];
  Flatten[output1, 1]
  ]
```

This function creates a graphical representation of a square. It takes in the points defining a square as input and returns a Graphics object representing the square.

```
displaySquare[points_] :=
 Module[{p1, p4},
  p1 = points[[1]];
  p4 = points[[4]];
  Graphics[{Opacity[RandomReal[]], Rectangle[p1, p4]}]
  ]
```

There is the body of the iteration method:

```
lengthList = {};
x = 1.0;
While[x > 0.01,
  AppendTo[lengthList, x];
  x = x/2;
  ];
  listOutput = {};
listCandidates =
  Table[{{i, j}, {i + 1, j}, {i, j + 1}, {i + 1, j + 1}}, {i, -5, 5,
     1}, {j, -5, 5, 1}] // Flatten[#, 1] &;
Table[
  (*finding square inside polygon *)
  selectedSquares =
   Cases[listCandidates, square_ /; testSquare[square, D1]];
  listOutput = Join[listOutput, selectedSquares];

  (*removing square inside polygon *)
  listCandidates = Complement[listCandidates, selectedSquares];
  (*rplitting remaining squares *)
  listCandidates = splitAllSquares[listCandidates, len];
  Print[len];
  Print[listOutput];
```

```
  , {len, lengthList[[1 ;; 4]]}];
plotRange = {{0, 3}, {0, 3}};
g1 = Graphics[Line[D1], PlotRange -> plotRange];
Show[
 g1,
 displaySquare /@ listOutput,
 PlotRange -> plotRange,
 Axes -> False
 ]
```

The output of the above codes is shown in Figure 1.1.1.

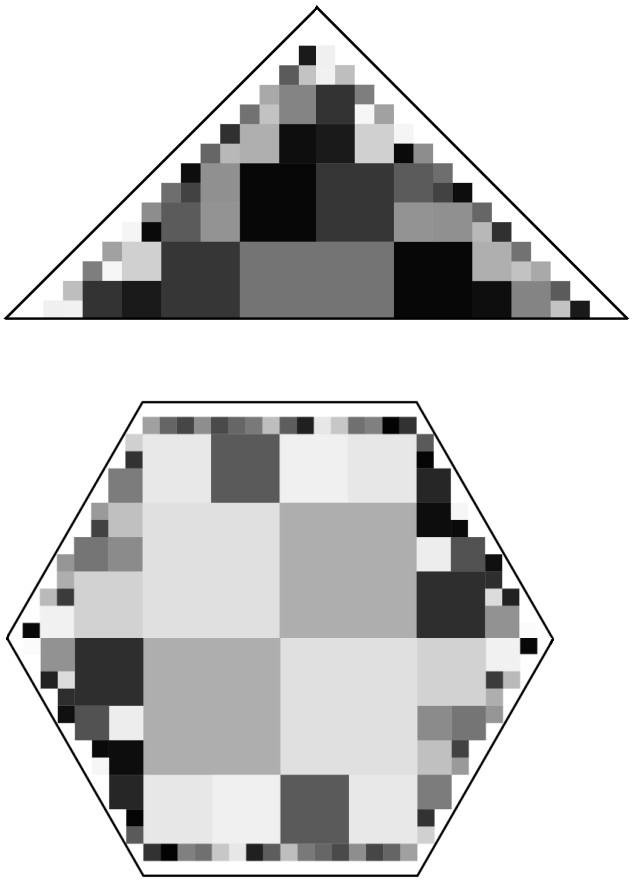


Figure 3.1.1: Examples of Whitney decomposition on 2D polygons

3.2 3D Generalization of Whitney Decomposition

Extending the Whitney decomposition from 2D to 3D requires the introduction of cubes instead of squares. The principles remain the same: we aim to approximate a bounded 3D domain (or polyhedron) using cubes of side length 2^{-k}, $k \in Z$.

In the Mathematica implementation of the 3d cases, a built-in function RegionMember is used to determine if a vertex is inside the polyhedron or not, for the sake of simplicity. The algorithm is very similar to the 2D case.

First, we define a cuboid as our region of interest:

```
region=ImplicitRegion[{0<=x<=8,0<=y<=4,0<=z<=8},{x,y,z}];
DiscretizeRegion[region]
```

This code generates an implicit region that represents the interior of the sphere.

Next, we define a function cubeSplit that splits a given cube into smaller cells. The function takes in a pair of points (the opposite corners of the cube) and returns the corners of the new smaller cells:

```
Clear[cubeSplit]
cubeSplit[{p0_, p1_}] :=
 Module[{center, edge, newbasecube, baselayer, toplayer},
  center = Mean[{p0, p1}];
  edge = EuclideanDistance[p1, p0]/Sqrt[3];
  newbasecube = {p0, center};
  baselayer =
   SortBy[Norm] /@
    Table[RotationTransform[n Pi/2, {0, 0, 1}, center] /@
      newbasecube, {n, 0, 3}];
  toplayer =
   baselayer /. {x_, y_, z_} -> ({x, y, z} + {0, 0, Sign[(p1 - p0)[[3]]]
      edge/2});
  Flatten[{baselayer, toplayer}, 1]]
```

This function refines the cube into smaller cubes if it partially resides inside the region. The cube is discarded if it is completely outside the region. If all vertices of the cube reside within the region and its volume is less than the maximum cell volume, the cube is considered as adequately refined:

```
Clear[refineCube]
refineCube[{p0_, p1_}, regionmemberfun_, maxvol_] :=
 Module[{vertices, verteval},(*generate all vertices*)
  vertices =
   Table[RotationTransform[n Pi/2, {0, 0, 1}, Mean[{p0, p1}]] /@ {p0,
     p1}, {n, 0, 3}];
  verteval = Flatten@regionmemberfun@vertices;
  Which[(*all False:cube must be completely out of region;drop it*)
   Nor @@ verteval, Nothing,
   And @@ verteval, {p0, p1},
   True,
   If[Abs[Times @@ (p1 - p0)] > maxvol,
    Sequence @@ cubeSplit[{p0, p1}], {p0, p1}]]]
```

Finally, we apply the refinement function iteratively until all the cubes are adequately refined. The result is a set of cells that approximates the shape of the sphere.By manipulating the maximum cell volume, we can control the granularity of the decomposition.

```
maxCellVolume = 0.05;
cubemeshpoints =
  FixedPoint[
   Function[{cube}, Map[refineCube[N@#, rmf, maxCellVolume] &, cube]],
    cubeSplit@{{-5, -5, -5}, {5, 5, 5}}];
    Graphics3D[{Yellow, Opacity[0.2], Cuboid @@@ cubemeshpoints},
 Axes -> True, Lighting -> "Neutral"]
```

The output of the above codes is the following:

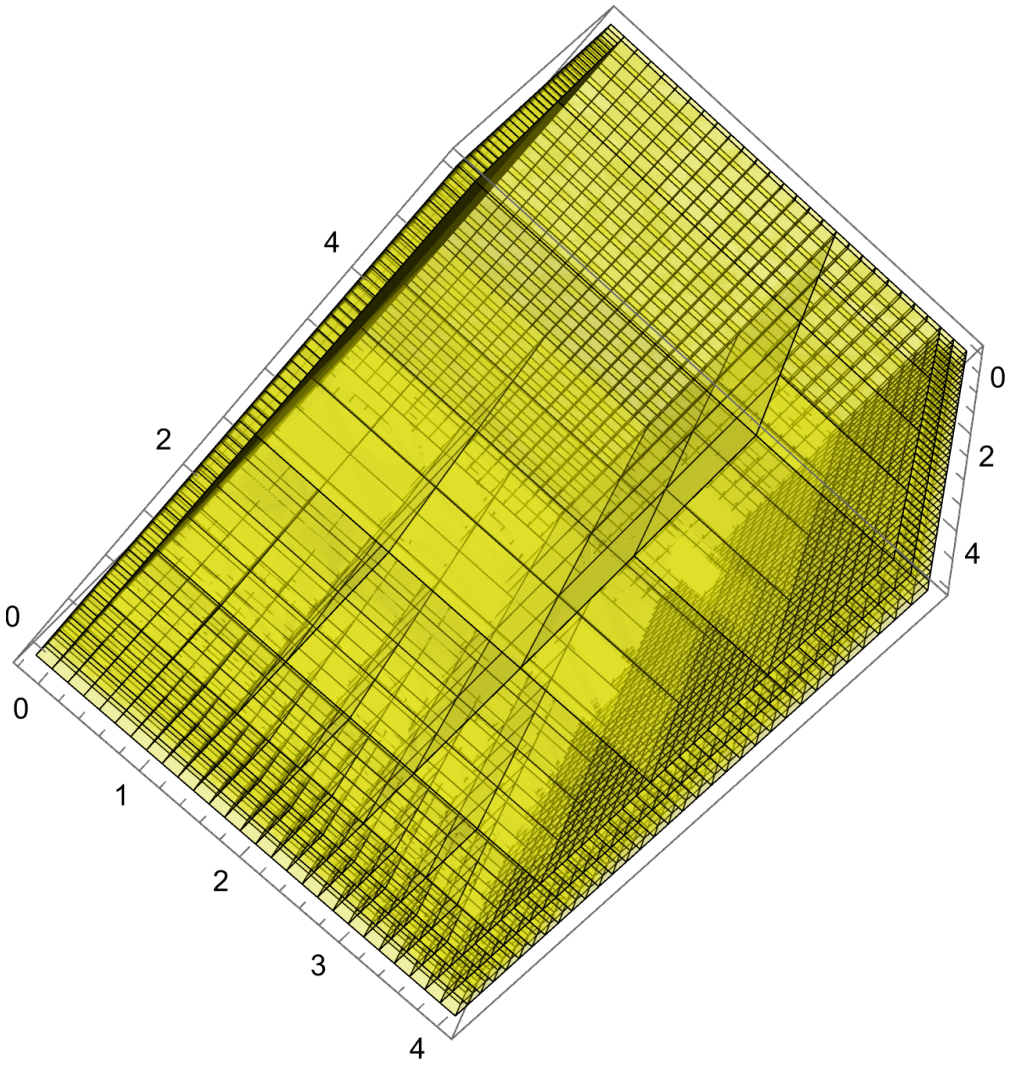


Figure 3.2.1: Whitney decomposition on a cuboid

4 Summary

This research outlines an investigation into the generalization of Whitney decomposition in three dimensions, The primary focus of the research is to expand on the existing knowledge of two-dimensional Whitney decomposition and extend it to three-dimensional domains.

Whitney decomposition enables us to better understand complex geometric structures by breaking them down into simpler, more manageable pieces. The paper presents new mathematical techniques for applying this decomposition in two and three dimensions, and these techniques may open up new opportunities for research in geometric analysis.

References

[1] R Klén P Hariri and M Vuorinen. "Whitney Decomposition". In: *Conformally Invariant Metrics and Quasiconformal Mappings*. Springer Nature Switzerland AG, 2020, pp. 92–93. DOI: 10.1007/978-3-030-32068-3. URL: https://link.springer.com/book/10.1007/978-3-030-32068-3.

[2] W. R. Franklin. *Personal website*. https://wrfranklin.org/Research/Short_Notes/pnpoly.html.

[3] D. Marshall. *Zipper software package for numerical conformal mappings*. https://sites.math.washington.edu/ marshall/zipper.html.