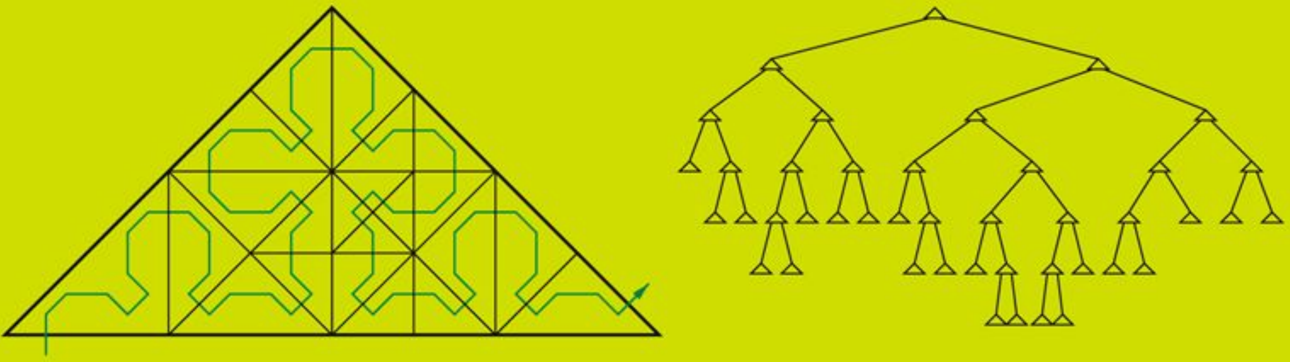


TEXTS IN COMPUTATIONAL SCIENCE
AND ENGINEERING

9

Michael Bader

Space-Filling Curves

An Introduction with Applications in Scientific Computing

Editorial Board
T. J. Barth
M. Griebel
D. E. Keyes
R. M. Nieminen
D. Roose
T. Schlick

Springer

Texts in Computational Science and Engineering

9

Editors

Timothy J. Barth
Michael Griebel
David E. Keyes
Risto M. Nieminen
Dirk Roose
Tamar Schlick

For further volumes:
http://www.springer.com/series/5151

Michael Bader

Space-Filling Curves

An Introduction with Applications in Scientific Computing

Springer

Michael Bader
Department of Informatics
Technische Universität München
Germany

ISSN 1611-0994
ISBN 978-3-642-31045-4 ISBN 978-3-642-31046-1 (eBook)
DOI 10.1007/978-3-642-31046-1
Springer Heidelberg New York Dordrecht London

Library of Congress Control Number: 2012949128

Mathematics Subject Classification (2010): 68W01, 14H50, 65Y05, 65Y99, 68U99, 90C99

© Springer-Verlag Berlin Heidelberg 2013
This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed. Exempted from this legal reservation are brief excerpts in connection with reviews or scholarly analysis or material supplied specifically for the purpose of being entered and executed on a computer system, for exclusive use by the purchaser of the work. Duplication of this publication or parts thereof is permitted only under the provisions of the Copyright Law of the Publisher's location, in its current version, and permission for use must always be obtained from Springer. Permissions for use may be obtained through RightsLink at the Copyright Clearance Center. Violations are liable to prosecution under the respective Copyright Law.
The use of general descriptive names, registered names, trademarks, service marks, etc. in this publication does not imply, even in the absence of a specific statement, that such names are exempt from the relevant protective laws and regulations and therefore free for general use.
While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Printed on acid-free paper

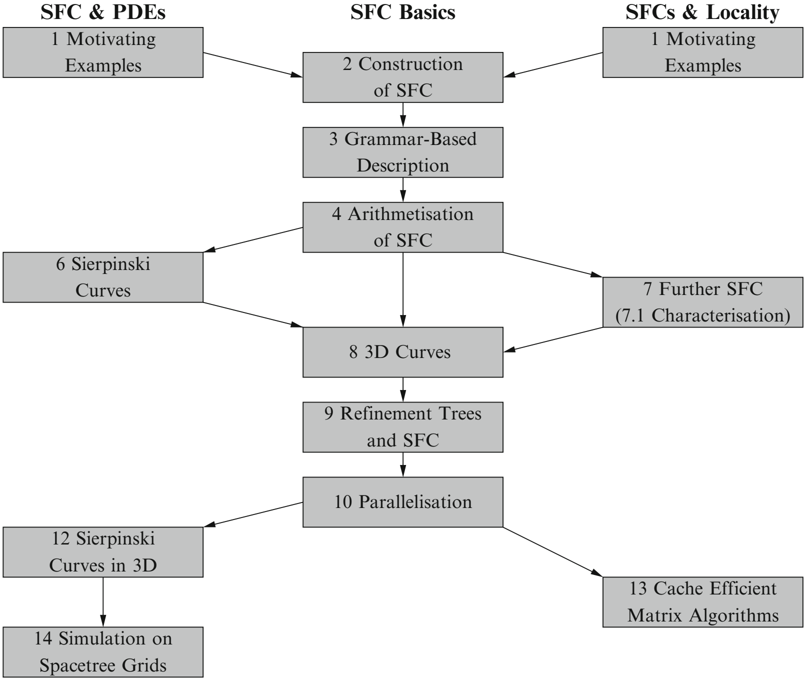
Springer is part of Springer Science+Business Media (www.springer.com)

Preface

Space-filling curves started their “lives” as mathematical curiosities, at the end of the nineteenth century. The idea that a one-dimensional curve may completely cover an area or a volume was, at that time, completely novel and counter-intuitive. Together with their relatives – such as Koch curves, the Cantor Set, and similar constructions – space-filling curves contradicted the then existing notion of a curve and of continuity and differentiability of functions (being continuous everywhere, but nowhere differentiable), and demanded new concepts in set theory and dimensionality. As a result, space-filling curves became almost notorious as “topological monsters”, and were studied by a good number of highly influential mathematicians, such as Peano, Hilbert, Lebesgue, Sierpinski, and others. The book of Hans Sagan [233] provides an excellent introduction and overview of these mathematical aspects of space-filling curves, as well as on their history.

The recursive and self-similar construction of space-filling curves leads to important locality properties – to put it in a nutshell, a space-filling-curve mapping will ensure that close-by parameters will be mapped to neighbouring points in the target set and – at least to a good extent – vice versa. It turned out that these properties are highly useful in the computational sciences. Algorithms based on space-filling curves can thus be used to construct spatial partitions to distribute a problem to different processors of a parallel computer, to improve the memory access behaviour of algorithms, or in general to find efficient data structures for multidimensional data. Most of the time, space-filling curves are not necessarily the best algorithm for the job where they are used – however, as Bartholdi and Platzman [32] put it, they are like “a good pocketknife”, being “simple and widely applicable”, and able to provide a convenient and satisfactory solution with comparably small effort in implementation and computation.

The aim of this book is therefore to give an introduction to the *algorithmics* of space-filling curves – to the various ways of describing them, and how these different description techniques lead to algorithms for different computational tasks. The focus will be on algorithms and applications in scientific computing, with a certain preference on mesh-based methods for partial differential equations.


SFC & PDEs
SFC Basics
SFCs & Locality
1 Motivating Examples
2 Construction of SFC
1 Motivating Examples
3 Grammar-Based Description
4 Arithmetisation of SFC
6 Sierpinski Curves
7 Further SFC (7.1 Characterisation)
8 3D Curves
9 Refinement Trees and SFC
10 Parallelisation
12 Sierpinski Curves in 3D
13 Cache Efficient Matrix Algorithms
14 Simulation on Spacetree Grids

How to Read This Book

The present book will hopefully serve multiple purposes – as a monograph on space-filling curves; as a reference to look up specific aspects, algorithms, or techniques to describe a specific curve; or as a textbook to be used as supplementary material for a course related to scientific computing, or even for a series of lectures that is dedicated to space-filling curves in particular. As a consequence, the sequential order of chapters I chose for this book will necessarily not be able to match the needs of every reader or lecturer. You are thus encouraged to read through this book in a non-sequential way, skip chapters or sections, do detours to later chapters, or similar. Some suggestions for selecting chapters and placing a different focus during a course ("just the basics" vs. techniques for partial differential equations vs. special focus on locality properties) are given in the figure above. In addition, every chapter will end with a box called "What's next?" that will give some suggestions on where to read on.

Additional material (solution to exercises, code examples, links to other material, errata if necessary) will be published on the website

`www.space-filling-curves.org`.

Acknowledgements

The topic of space-filling curves has been a part of my research and teaching work for the last 8 years, at least, and numerous colleagues and fellow researchers provided valuable input and ideas. I would especially like to thank all my colleagues at Technische Universität München and at Universität Stuttgart – while I cannot list all of them, I want to express my special gratitude to those who have been my colleagues for the longest time: Hans Bungartz, Miriam Mehl, Tobias Neckel, Tobias Weinzierl, and Stefan Zimmer. Amongst all colleagues from other research groups, I would like to give thanks and tribute to Jörn Behrens, Herman Haverkort, Bill Mitchell, and Gerhard Zumbusch, who provided lots of inspiration for this book – especially for the chapters on Sierpinski curves and on locality properties.

Finally, and most of all, I want to thank Christoph Zenger, my former academic supervisor and mentor. In 2003, we initiated a joint lecture on *Algorithms in Scientific Computing*, with space-filling curves being one of three major topics. It was the first course, in which I was solely responsible for the content of a lecture, it has been running (with certain topical updates, of course) in this form for 8 years, and it was the starting-point and foundation for this book project.

Munich April 23, 2012 Michael Bader

Contents

1 Two Motivating Examples: Sequential Orders on Quadtrees and Multidimensional Data Structures 1
1.1 Modelling Complicated Geometries with Quadtrees, Octrees, and Spacetrees 1
1.1.1 Quadtrees and Octrees 3
1.1.2 A Sequential Order on Quadtree Cells 3
1.1.3 A More Local Sequential Order on Quadtree Cells 6
1.2 Numerical Simulation: Solving a Simple Heat Equation 7
1.3 Sequentialisation of Multidimensional Data 9
1.3.1 Requirements for Efficient Sequential Orders 11
1.3.2 Row-Major and Column-Major Sequentialisation 12

2 How to Construct Space-Filling Curves 15
2.1 Towards a Bijective Mapping of the Unit Interval to the Unit Square 15
2.2 Continuous Mappings and (Space-Filling) Curves 17
2.3 The Hilbert Curve 18
2.3.1 Iterations of the Hilbert Curve 18
2.3.2 Approximating Polygons 19
2.3.3 Definition of the Hilbert Curve 20
2.3.4 Proof: *h* Defines a Space-Filling Curve 22
2.3.5 Continuity of the Hilbert Curve 23
2.3.6 Moore's Version of the Hilbert Curve 24
2.4 Peano Curve 25
2.5 Space-Filling Curves: Required Algorithms 27

3 Grammar-Based Description of Space-Filling Curves 31
3.1 Description of the Hilbert Curve Using Grammars 31
3.2 A Traversal Algorithm for 2D Data 34
3.3 Grammar-Based Description of the Peano Curve 37
3.4 A Grammar for Turtle Graphics 39

4 Arithmetic Representation of Space-Filling Curves 47
4.1 Arithmetic Representation of the Hilbert Mapping 47
4.2 Calculating the Values of h 49
4.3 Uniqueness of the Hilbert Mapping 52
4.4 Computation of the Inverse: Hilbert Indices 55
4.5 Arithmetisation of the Peano Curve 57
4.6 Efficient Computation of Space-Filling Mappings 59
4.6.1 Computing Hilbert Mappings via Recursion Unrolling 60
4.6.2 From Recursion Unrolling to State Diagrams 61

5 Approximating Polygons 67
5.1 Approximating Polygons of the Hilbert and Peano Curve 67
5.2 Measuring Curve Lengths with Approximating Polygons 69
5.3 Fractal Curves and Their Length 70
5.4 A Quick Excursion on Fractal Curves 72

6 Sierpinski Curves 77
6.1 The Sierpinski-Knopp Curve 77
6.1.1 Construction of the Sierpinski Curve 77
6.1.2 Grammar-Based Description of the Sierpinski Curve 79
6.1.3 Arithmetisation of the Sierpinski Curve 80
6.1.4 Computation of the Sierpinski Mapping 81
6.2 Generalised Sierpinski Curves 82
6.2.1 Bisecting Triangles Along Tagged Edges 83
6.2.2 Continuity and Locality of Generalised Sierpinski Curves 85
6.2.3 Filling Triangles with Curved Edges 87

7 Further Space-Filling Curves 93
7.1 Characterisation of Space-Filling Curves 93
7.2 Lebesgue Curve and Morton Order 95
7.3 The H-Index 99
7.4 The $\beta\Omega$-Curve 101
7.5 The Gosper Flowsnake 104

8 Space-Filling Curves in 3D 109
8.1 3D Hilbert Curves 109
8.1.1 Possibilities to Construct a 3D Hilbert Curve 109
8.1.2 Arithmetisation of the 3D Hilbert Curve 113
8.1.3 A 3D Hilbert Grammar with Minimal Number of Non-Terminals 114
8.2 3D Peano Curves 116
8.2.1 A Dimension-Recursive Grammar to Construct a 2D Peano Curve 116
8.2.2 Extension of the Dimension-Recursive Grammar to Construct 3D Peano Curves 117

8.2.3 Peano Curves Based on 5 × 5 or 7 × 7 Refinement 119
8.2.4 Towards Peano's Original Construction 122
8.3 A 3D Sierpinski Curve 123

9 Refinement Trees and Space-Filling Curves 129
9.1 Spacetrees and Refinement Trees 129
9.1.1 Number of Grid Cells for the Norm Cell Scheme and for a Quadtree 131
9.2 Using Space-Filling Curves to Sequentialise Spacetree Grids 132
9.2.1 Adaptively Refined Spacetrees 134
9.2.2 A Grammar for Adaptive Hilbert Orders 135
9.2.3 Refinement Information as Bitstreams 137
9.3 Sequentialisation of Adaptive Grids Using Space-Filling Curves 138

10 Parallelisation with Space-Filling Curves 143
10.1 Parallel Computation of the Heat Distribution on a Metal Plate 143
10.2 Partitioning with Space-Filling Curves 146
10.3 Partitioning and Load-Balancing Based on Refinement Trees and Space-Filling Curves 149
10.4 Subtree-Based Load Distribution 150
10.5 Partitioning on Sequentialised Refinement Trees 153
10.5.1 Modified Depth-First Traversals for Parallelisation 153
10.5.2 Refinement Trees for Parallel Grid Partitions 155
10.6 Data Exchange Between Partitions Defined via Space-Filling Curves 157
10.6.1 Refinement-Tree Partitions Using Ghost Cells 159
10.6.2 Non-Overlapping Refinement-Tree Partitions 160

11 Locality Properties of Space-Filling Curves 167
11.1 Hölder Continuity of Space-Filling Curves 167
11.1.1 Hölder Continuity of the 3D Hilbert Curve 168
11.1.2 Hölder Continuity and Parallelisation 168
11.1.3 Discrete Locality Measures for Iterations of Space-Filling Curves 171
11.2 Graph-Related Locality Measures 172
11.2.1 The Edge Cut and the Surface of Partition Boundaries 173
11.2.2 Connectedness of Partitions 174

12 Sierpinski Curves on Triangular and Tetrahedral Meshes 181
12.1 Triangular Meshes and Quasi-Sierpinski Curves 181
12.1.1 Triangular Meshes Using Red–Green Refinement 181
12.1.2 Two-Dimensional Quasi-Sierpinski Curves 182
12.1.3 Red–Green Closure for Quasi-Sierpinski Orders......... 184

12.2 Tetrahedral Grids and 3D Sierpinski Curves 184
12.2.1 Bisection-Based Tetrahedral Grids 184
12.2.2 Space-Filling Orders on Tetrahedral Meshes 188

13 Case Study: Cache Efficient Algorithms for Matrix Operations 195
13.1 Cache Efficient Algorithms and Locality Properties 195
13.2 Cache Oblivious Matrix-Vector Multiplication 199
13.3 Matrix Multiplication Using Peano Curves 201
13.3.1 Block-Recursive Peano Matrix Multiplication 204
13.3.2 Memory Access Patterns During the Peano Matrix Multiplication 205
13.3.3 Locality Properties and Cache Efficiency 206
13.3.4 Cache Misses on an Ideal Cache 208
13.3.5 Multiplying Matrices of Arbitrary Size 210
13.4 Sparse Matrices and Space-Filling Curves 211

14 Case Study: Numerical Simulation on Spacetree Grids Using Space-Filling Curves 215
14.1 Cache-Oblivious Algorithms for Element-Oriented Traversals ... 215
14.1.1 Element-Based Traversals on Spacetree Grids 217
14.1.2 Towards Stack-Based Traversals 219
14.2 Implementation of the Element-Oriented Traversals 221
14.2.1 Grammars for Stack Colouring 222
14.2.2 Input/Output Stacks Versus Colour Stacks 222
14.2.3 Algorithm for Sierpinski Traversal 225
14.2.4 Adaptivity: An Algorithm for Conforming Refinement 226
14.2.5 A Memory-Efficient Simulation Approach for Dynamically Adaptive Grids 227
14.3 Where It Works: And Where It Doesn't 228
14.3.1 Three-Dimensional Hilbert Traversals 229
14.3.2 A Look at Morton Order 229

15 Further Applications of Space-Filling Curves: References and Readings 235

A Solutions to Selected Exercises 239
A.1 Two Motivating Examples 239
A.2 Space-Filling Curves 239
A.3 Grammar-Based Description of Space-Filling Curves 240
A.4 Arithmetic Representation of Space-Filling Curves 241
A.5 Approximating Polygons 243
A.6 Sierpinski Curves 246
A.7 Further Space-Filling Curves 246
A.8 Space-Filling Curves in 3D 247
A.9 Refinement Trees and Space-Filling Curves 248

A.10 Parallelisation with Space-Filling Curves 249
A.11 Locality Properties of Space-Filling Curves 251
A.12 Sierpinski Curves on Triangular and Tetrahedral Meshes 251
A.13 Cache Efficient Algorithms for Matrix Operations 253
A.14 Numerical Simulation on Spacetree Grids Using Space-Filling Curves 253

References 257

Index 271

Chapter 1
Two Motivating Examples: Sequential Orders on Quadtrees and Multidimensional Data Structures

In Scientific Computing, space-filling curves are quite commonly used as tools to improve certain properties of data structures or algorithms, or even to provide or simplify algorithmic solutions to particular problems. In the first chapter we will pick out two simple examples – both related to data structures and algorithms on computational grids – to introduce typical data structures and related properties and problems that can be tackled by using space-filling curves.

1.1 Modelling Complicated Geometries with Quadtrees, Octrees, and Spacetrees

Techniques to efficiently describe and model the geometry of solid objects are an inherent part of any lecture or textbook on Computer Graphics. Important applications arise, for example, in *computer aided design* (CAD) – in the car industry this might be used to design car bodies, engine components, or even an entire motor vehicle. However, once the design is no longer our sole interest, but the structural stability of components, aerodynamics of car bodies, or similar questions are investigated that require numerical simulation or computational methods, the geometry modelling of such objects also becomes an important question in scientific computing.

In geometry modelling, we distinguish between *surface*- and *volume-oriented* models. Surface-oriented models first describe the topology of an object, via vertices, edges, and faces, their position and how they are connected. The most simple model of this kind is the so-called *wire-frame model* (see Fig. 1.1) – well-known from classical computer games and old-fashioned science-fiction movies. While wire-frame models are no longer much used in practice, their extensions enable us to model edges and faces via Bézier curves or surfaces, NURBS, or other higher-order curves and surfaces, and are still state-of-the-art in CAD. Hence,

M. Bader, *Space-Filling Curves*, Texts in Computational Science and Engineering 9,
DOI 10.1007/978-3-642-31046-1_1, © Springer-Verlag Berlin Heidelberg 2013

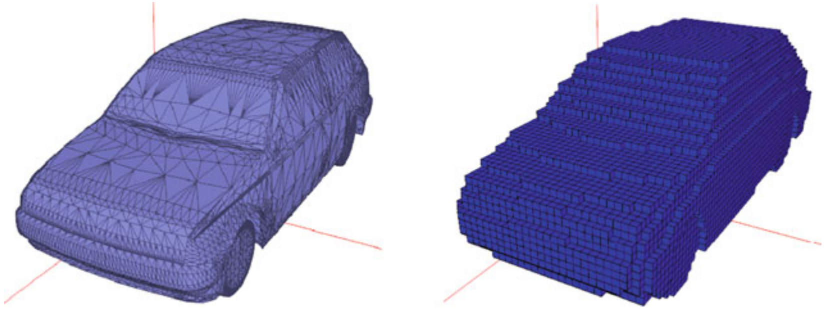


Fig. 1.1 3D geometry models of a car: surface-oriented modelling with a wire-frame model (*left image*) vs. a 2D norm-cell scheme (*right image*) (Images reproduced (with permission) from Mundani and Daubner [72])

the model describes the surface of an object, which separates the interior (the object itself) from the exterior of the object.

In contrast, volume-oriented models characterise an object directly via the space it occupies. The simplest volume-oriented model is the so-called *norm cell* scheme. There, the object to be described is embedded into a regular mesh of cuboid cells. For each of the cells, we then store whether it is inside or outside the object. The norm cell meshes correspond to Cartesian meshes, as they are frequently used for numerical discretisation in scientific computing. There, continuous functions are approximated on such meshes, and each cell (or cell node) contains an approximate function value or another approximation of the function (a low-order polynomial, e.g.). In that sense, the norm cell scheme would represent the characteristic function of an object, which is defined as 1 in the interior and 0 outside of the object.

For the classical surface-oriented models, the storage requirements grow with the complexity of the object. In contrast, the storage requirement of the norm cell scheme solely depends on the chosen resolution, i.e. the size of the norm cells. For each of the cells, the norm cell scheme requires at least one bit to represent whether a cell is inside or outside the domain. If different material properties are to be modelled, the storage requirement per norm cell will grow respectively. Let's compute a quick estimate of the typical amount of required memory. We assume a car body with an approximate length of 4 m, and width and height of 1.5 m each. Using a uniform resolution of 1 cm, we will already need $400 \times 150 \times 150$ grid cells – which is around nine million. Though this already puts our memory requirement into the megabyte range, it's clear that modelling a car as a conglomerate of 1 cm-bricks won't be sufficient for a lot of problems. For example, to decide whether the doors or boot panel of a car will open and close without collisions, no engineer would allow a tolerance of 1 cm. Nor could the aerodynamics of the car be computed too precisely. However, if we increase the resolution to 1 mm, the number of grid cells will rise by a factor of 10^3, and the required memory will move into the gigabyte range – which is no longer easily manageable for typical workstations.

1.1.1 Quadtrees and Octrees

The so-called *quadtrees* (in 2D) and *octrees* (in 3D) are geometrical description models that are designed to limit the rapid increase of the number of cells as is the case for the regularly refined norm cells. The main idea for the quadtree and octree models is to increase the resolution of the grid only in areas where this is necessary. This is achieved by a recursive approach:

1. We start with a grid with very coarse resolution – usually with a grid that consists of only a single cell that embeds the entire object.
2. Following a recursive process, we now refine the grid cells step by step until all grid cells are either inside or outside of the object, or else until a given cell size is reached, which is equal to the desired resolution.
3. During that process, cells that are entirely inside or outside the objects, are not refined any further.
4. All other cells – which contain at least a small part of the object boundary – are subdivided into smaller grid cells, unless the finest resolution is already reached.

By default, the grid cells are subdivided into congruent subcells. For example, to subdivide square cells into four subsquares of identical size is a straightforward choice in 2D and leads to the so-called *quadtrees*. Figure 1.2 illustrates the subsequent generation of such a recursively structured quadtree grid for a simple example object.

Tree structures would be a possible choice for a data structure that describes such grids: each grid cells corresponds to a node of the tree. The starting cell defines the root of the tree, and for each cell its four subsquares are the children of the respective node. For our 2D, square-based recursive grid, we obtain a tree where each node has exactly four children, which explains why the name *quadtree* is used for such recursively structured grids. Our example given in Fig. 1.2 shows the quadtree that corresponds to our recursively refined grid.

The quadtree-based grid generation can be easily extended to the 3D case. We then use cubic cells, which are subdivided into eight cubes of half size. In the corresponding tree structure, each node therefore has eight children, and we obtain a so-called *octree*. A respective 3D example is given in Fig. 1.3.

1.1.2 A Sequential Order on Quadtree Cells

Assume that we want to process or update the data stored in a quadtree in a specific way. We then have to perform a *traversal* of the quadtree cells, i.e. visit all quadtree cells at least once. The straightforward way to do this is to perform a traversal of the corresponding tree structure. In a so-called *depth-first* traversal this is achieved by recursively descending in the tree structure, always following the left-most unvisited branch, until a leaf cell is reached. After processing this leaf cell, the traversal steps

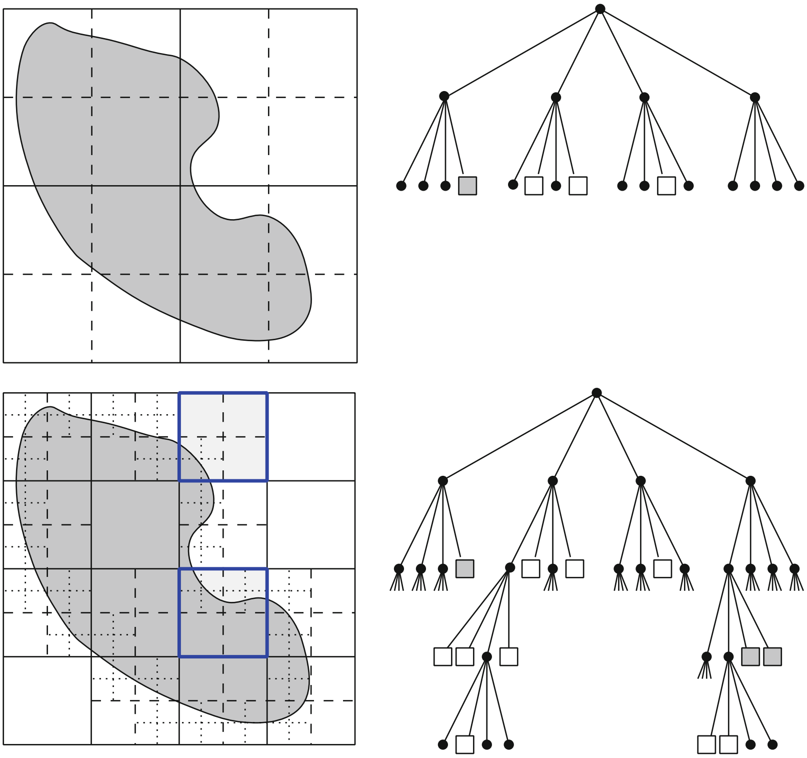


Fig. 1.2 Generating the quadtree representation of a 2D domain. The quadtree is fully extended only for the highlighted cells

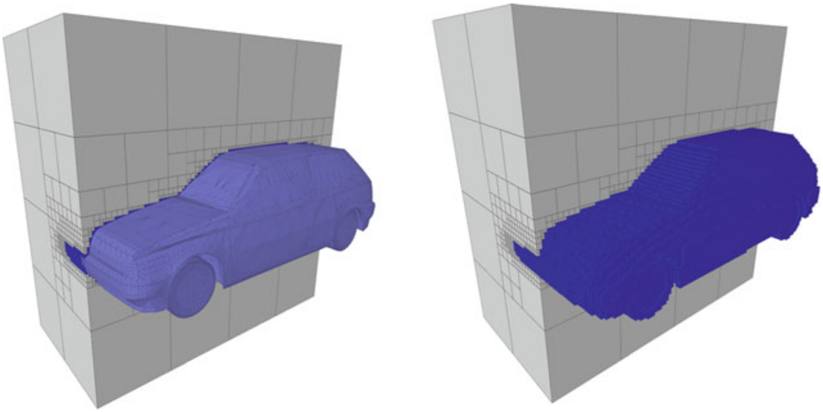


Fig. 1.3 Generating the octree representation of a car body from a respective surface model (Images with permission from Mundani and Daubner [72])

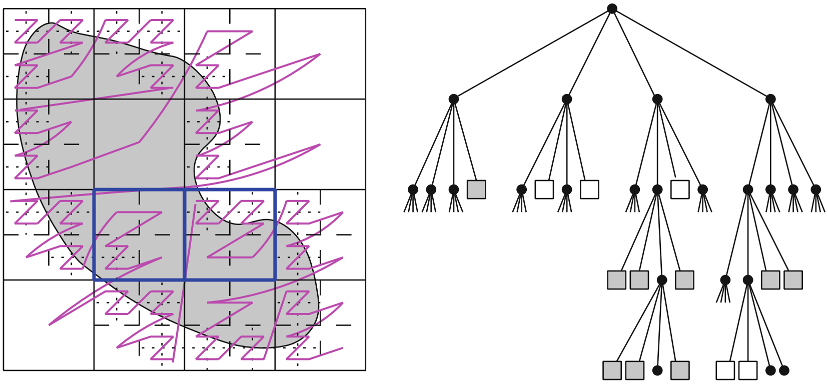


Fig. 1.4 Quadtree representation of a given object and a simple sequential order on the quadtree cells that results from a depth-first traversal of the corresponding tree structure

back up the tree until the first node is reached that still has a child node that was not visited throughout the traversal. A recursive implementation of this depth-first traversal is given in Algorithm 1.1. Note that the call tree of this algorithm has the same structure as the quadtree itself.

Algorithm 1.1: Depth-first traversal of a quadtree

```
Procedure DFtraversal(node)
    Parameter: node: current node of the quadtree (root node at entry)
begin
    if isLeaf(node) then
        // process current leaf node
    else
        foreach child ∈ leaves(node) do
            DFtraversal(child);
        end
    end
end
```

In what order will the cells of the quadtree be visited by this depth-first traversal of the tree? This, of course, depends on the order in which we process the children or each node in the **foreach**-loop. In the tree, we may assume that child nodes are processed from left to right, but this does not yet fix the spatial position of the four subcells. Assuming a local row-wise order – top-left, top-right, bottom-left, bottom-right – in each node, we finally obtain a sequential order of the leaf cells of the quadtree. This order is illustrated in Fig. 1.4.

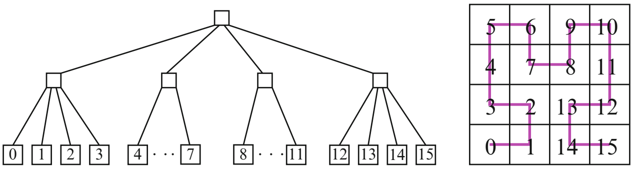


Fig. 1.5 Quadtree representation of a regular 4 × 4 grid, and a sequential order that avoids jumps

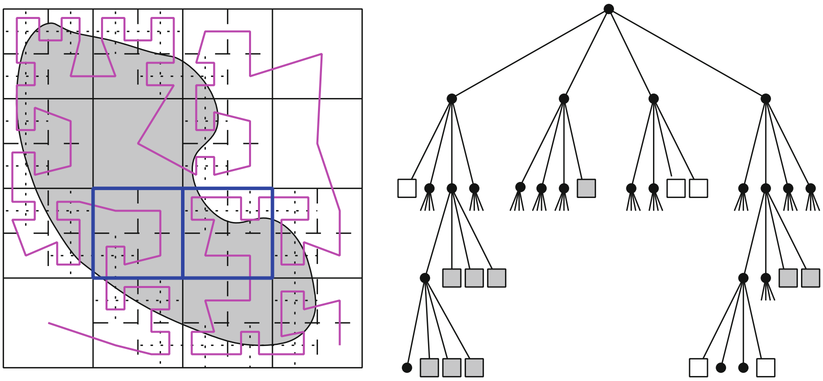


Fig. 1.6 Quadtree representation of a given object and a sequential order on the quadtree cells that avoids jumps – the order corresponds to a depth-first traversal of the corresponding tree

1.1.3 A More Local Sequential Order on Quadtree Cells

We observe that the generated order leads to pretty large jumps between one cell and its successor in the sequential order. Is it possible to generate an order where successor and predecessor are also direct neighbours of a cell, i.e. share a common boundary? For a comparably simple quadtree grid, such as a regular 4 × 4-grid, it is not too difficult to come up with such a sequential order. Figure 1.5 shows an example. The respective numbering scheme can be extended to larger grids and finally leads to the *Hilbert curve*, which we will introduce in Chap. 2.

We can also adopt this Hilbert order to sequentialise the cells of our quadtree grid, as illustrated in Fig. 1.6. Jumps are totally avoided in this sequential order, as two cells that are neighbours according to the sequential order are also geometrical neighbours in the sense that they have a common boundary. Sequential orders that are able to maintain such neighbour properties are useful in a lot of applications. One of those applications is the definition of equal-sized and compact partitions for parallelisation of such quadtree grids. We will discuss such partitioning approaches in Chaps. 9 and 10. We will show that the new sequentialisation again corresponds to a depth-first traversal of the corresponding tree structure. However, the tree structure for the new order differs from that used in Fig. 1.4 and uses a different local order of the children of each node.

While we might be able to draw such a sequential order into a given quadtree, we do not have the required mathematical or computational tools available, yet, to describe the generated order in a precise way, or to give an algorithm to generate it. As for the simpler order adopted in Fig. 1.4, the order results from a local ordering of the four children of each node of the quadtree. However, as we can observe from Fig. 1.6, that local order is no longer uniform for each node. We will discuss a grammar-based approach to define these local orders in Chap. 3, and use it to derive traversal and other algorithms for quadtrees in Chaps. 9 and 14.

1.2 Numerical Simulation: Solving a Simple Heat Equation

Let us move to a further example, which is motivated by the field of numerical simulation. As a simple example for such a problem, we examine the computation of the temperature distribution on a metal plate. The geometrical form of the metal plate shall be given, which also defines the computational domain Ω of our problem. Moreover, we assume that the temperature is known at the boundaries of the plate, and that we know all heat sources and heat drains in the interior of the plate. We further assume that, with the exception of these heat sources and drains, the plate can only lose or gain heat energy across the boundaries of the plate. Finally, all boundary conditions, as well as the heat sources and heat drains are assumed to be constant in time, such that we can expect a *stationary* temperature distribution on the metal plate. This temperature equilibrium is what we try to compute.

To be able to solve this problem on a computer, we first require a respective mathematical model. Typically, we define a grid of measurement point on the metal plate. We will compute the temperature only at these mesh points. Figure 1.7 shows two examples of such grids. The one on the left is a uniform, rectangular grid (so-called *Cartesian* grid) that is defined on a square metal plate. The example in the right image shows an unstructured, triangular grid for a metal plate that has a more complicated geometry. Of course, we could also use a quadtree- or octree-based grid, as discussed in the previous section.

In the following, we will assume that the grid points will hold our measurement points for the temperature, i.e. the grid points determine the position of our unknowns.

The approximation to compute the temperature on the grid points, only, replaces our problem of finding a continuous temperature distribution by the much simpler problem (at least for a computer) of computing the temperature at a finite number of grid points. To determine these temperatures, we will set up a system of equations for the respective temperatures. In a “serious” application, the respective system of equations would be derived from the discretisation of a partial differential equation for the temperature. However, in the present example, we can get by with a much simpler model.

From physics we know that, without heat sources or drains, the equilibrium temperature at a grid point will be the average of the temperatures at the neighbouring

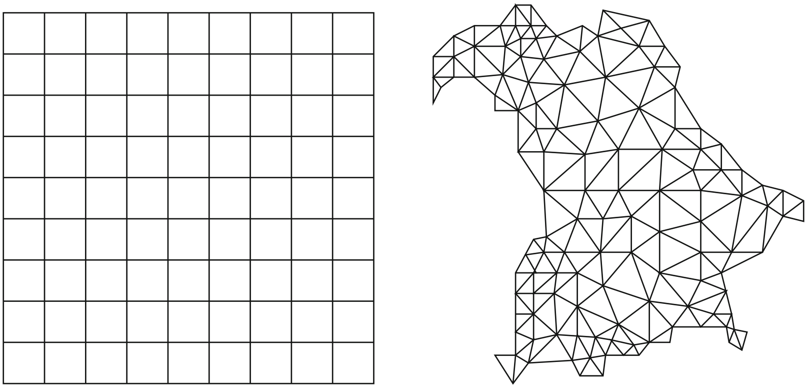

(a) Cartesian Grid (b) Triangular Grid

Fig. 1.7 Two examples for a computational grid: a uniform, *Cartesian* grid (*left image*) vs. an unstructured triangular grid on a complicated computational domain (*right image*). (**a**) Cartesian grid. (**b**) Triangular grid

grid points. In the computational grids given in Fig. 1.7, we could use the adjacent grid points, i.e. the grid point connected by grid lines, for this averaging.

In the Cartesian grid, we can also assume that the arithmetic average of the four adjacent grid points will lead to a good approximation of the real equilibrium situation. Hence, if we use the unknowns $u_{i,j}$ to denote the temperature at the grid point in the i-th row and j-th column of the Cartesian grid, we obtain the following equilibrium equation

$$u_{i,j} = \frac{1}{4}\left(u_{i-1,j} + u_{i,j-1} + u_{i+1,j} + u_{i,j+1}\right)$$

for each i and j. We can reformulate these equations into the standardised form

$$u_{i,j} - \frac{1}{4}\left(u_{i-1,j} + u_{i,j-1} + u_{i+1,j} + u_{i,j+1}\right) = 0.$$

Heat sources or drain may then be modelled by an additional right-hand side $f_{i,j}$:

$$u_{i,j} - \frac{1}{4}\left(u_{i-1,j} + u_{i,j-1} + u_{i+1,j} + u_{i,j+1}\right) = f_{i,j} \qquad \text{for all } i, j. \tag{1.1}$$

The solution of this system of linear equations will give us an approximation for the temperature distribution in equilibrium. The more grid points we will invest, the more accurate we expect our solution. Figure 1.8 plots the solution for a 16×16 grid. For that problem, no heat sources or drains were allowed, and zero temperature was assumed at three of the four boundaries. At the fourth boundary, a sine function was used to describe the temperature.

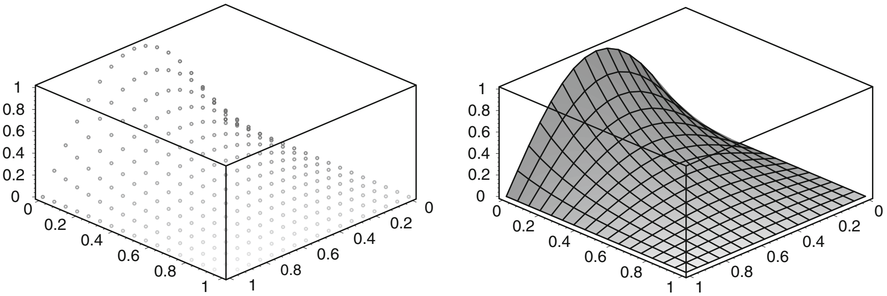


Fig. 1.8 Solution of our model problem on a 16 × 16 Cartesian grid. The *left plot* shows the temperatures at the grid points, whereas the *right plot* gives a bilinear interpolation of this pointwise solution

Towards Large-Scale Simulations

For a full-featured, large-scale simulation of realistic applications, we have to improve our simple computational model by a multitude of "features":

- We will require a large number of unknowns, which we might have to place *adaptively* in the computational grid, i.e. refine the grid only in regions where we require a more accurate solution.
- The large number of unknowns might force us to use sophisticated solvers for the resulting systems of equations, such as *multigrid methods*, which will use hierarchies of grids with different solutions to obtain faster solvers.
- Expensive computations or huge memory requirements for the large amount of unknowns will force us to use parallel computers or even supercomputers to compute the solution. We will have to *partition* our grid into parts of equal computational effort and allocate these partitions to different processors. We will need to take care that the *load distribution* between the processors is uniform, and perform *load balancing*, if necessary.

It turns out that for many of these problems, we need to *sequentialise* our grids in a certain way. Adaptive, multidimensional grids need to be stored in memory, which consists of a sequence of memory cells. We need loops to update all unknowns of a grid, which is again often done in sequential order. Finally, we will see that also parallelisation becomes simpler, if we can map our higher-dimensional grid to a simple sequence of grid points. Thus, the sequentialisation of multidimensional data will be our key topic in this book. And, again, we will introduce and discuss sequential orders for this purpose, such as illustrated in Figs. 1.4 and 1.6.

1.3 Sequentialisation of Multidimensional Data

The data structures used for our two examples are not the only ones that fall into the category of multidimensional data. In scientific computing, but also in computer science, in general, such multidimensional data structures are ubiquitous:

- Discretisation-based data and geometric models, such as in our two previous examples;
- Vectors, matrices, tensors, etc. in linear algebra;
- All kinds of (rasterised) image data, such as pixel-based image data from computer tomography, but also movies and animations;
- Coordinates, in general (often as part of graphs);
- Tables, as for example in data bases;
- Statistical data – in computational finance, for example, baskets of different stocks or options might be considered as multidimensional data.

Throughout this book, we will consider such multidimensional data, in general, i.e. all data sets where an n-tuple $(i_1, i_2, \ldots, i_n)$ of indices is mapped to corresponding data.

For the simple case of multidimensional arrays, more or less every programming language will offer respective data structures. Interestingly, already for the simple case of a 2D array, the implementation is not uniform throughout the various languages. While Fortran, for example, uses a column-major scheme (i.e. a column-wise, sequential order), Pascal and C opted for a row-wise scheme, whereas C and Java will also use a 1D array of pointers that point to the sequentialised rows. These different choices result from the problem that no particular data structure can be optimal for all possible applications.

Examples of Algorithm and Operations on Multidimensional Data

For the types of multidimensional data, as listed above, we can identify the following list of typical algorithms and operations that work on this data:

- Matrix operations (multiplication, solving systems of equations, etc.) in linear algebra;
- Traversal of data, for example in order to update or modify each member of a given data set;
- Storing and retrieving multidimensional data sets, both in/from the computers main memory or in/from external storage;
- Selecting particular data items or data subsets, such as a particular section of an image;
- Partitioning of data, i.e. distributing the data set into several parts (of approximately the same size, e.g.), in particular for parallel processing, or for divide-and-conquer algorithms;
- Accessing neighbouring elements – as observed in our example of the temperature equation, where always five neighbours determine a row of the system of equations to be solved.

There is probably a lot of further examples, and, again, in many of these operations, the *sequentialisation* of the data set forms a central subproblem:

- Traversal of data is a straightforward example of sequentialisation;

- Similarly, storing and retrieving data in main memory or on an external storage require sequentialisation, because the respective memory models are one-dimensional;
- Introducing a sequential order on the data, maybe in combination with sorting the data according to this order, can simplify algorithms for subsequent, more complicated problems.

In the broadest sense, we can interpret any operation that builds on nested loops, such as

```
for $i = 1, \dots, n$ do
    for $j = 1, \dots, m$ do
        ...
```

or similar constructs, as a traversal of multidimensional data.

1.3.1 Requirements for Efficient Sequential Orders

After we have recognised the importance of introducing sequential orders on multidimensional data structures, we can ask what requirements we should pose with respect to the efficiency and suitability of such orders. Some of these properties will be strictly necessary to make sequential orders work at all, while other demands will be efficiency-driven.

Necessary Properties of Sequential Orders

A first, necessary property of a sequential order is that is generates a *unique index* for each data item. First of all, this property ensures that we can safely access (i.e. store and retrieve) this data item via its index. In addition, a traversal of the indices will lead to a traversal of the data set, where all data items are processed exactly once – no data item will be skipped and none will be processed twice. In a mathematical sense, the mapping between indices and data items should be *bijective*.

For a contiguous data set, the sequentialisation should also lead to a contiguous sequence of indices – ideally from 0 to $n - 1$ (or 1 to n). “Holes” in the index range are therefore forbidden. We can argue, of course, whether this requirement is rather required for efficiency than a necessity, as a small overhead of unused indices might be acceptable in practice. However, we will take it as a characteristic feature of sequentialisations that such holes in the index range do not occur.

Finally, we will need sequential orders on data sets of quite different extension – such as arrays of different size or dimension. Hence, our indexing scheme needs to be adjustable based on certain parameters, which means that we require a *family* of sequential orders.

Requirements Regarding Efficiency

The following requirements for efficient sequentialisations will result from the respective applications, but also will depend on the respective use case. Hence, for different applications, only some of the requirements might be of importance, and the relative importance of the different properties will vary with applications, as well.

- First of all, the mapping between data items and indices should be easy to compute – in particular, the computation must be fast enough such that the advantages of the sequential order are not destroyed by the additional computational effort.
- Neighbour relations should be retained. If two data items are neighbours in the multidimensional data set, their indices should be close together, as well, and vice versa. This property will always be important, if neighbouring data items are typically accessed at the same time – consider extraction of image sections, for example, or solving systems of equations similar to (1.1). Retaining neighbour relations will then conserve the *locality* properties of data.
- Two variants of preserving locality are characterised by the *continuity* of the mapping for sequentialisation and the *clustering property*. Continuity ensures that data items with successive indices will be direct neighbours in the multidimensional data space, as well. The clustering property is satisfied, if a data set defined via a limited index range will not have a large extension in either direction of the data space – in 3D, for example, an approximately ball-shaped data range would be desired.
- The sequentialisation should not favour or penalise certain dimensions. The requirement is not only a fairness criterion, but may be of particular importance, if a uniform runtime of an algorithm is important, for example in order to predict the runtime accurately.

1.3.2 Row-Major and Column-Major Sequentialisation

For 2D arrays, *row-major* or *column-major* schemes, i.e. the sequentialisation by rows or columns, are the most frequent variants. A pixel-based image, for example, could be stored row by row – from left to right in each row, and sequentialising the rows from top to bottom. To store matrices (or arrays), row-major or column-major storage is used in many programming languages. The index of the elements A_{ij} is then computed via

$$\text{address}(A_{ij}) = in + j \qquad \text{or} \qquad \text{address}(A_{ij}) = i + jm,$$

where n is the number of columns and m the number of rows.

Which Properties Are Satisfied by Row-Major and Column-Major Schemes?

The properties proposed in the previous section are only partially satisfied by the row- or column-major schemes:

- Without doubt, the sequentialisation is easy and fast to compute – which explains the widespread use of the schemes.
- Neighbour relations are only preserved in one of the dimensions – for a row-major schemes, only the neighbours within the same row will be adjacent in the index space, as well. Neighbours in the column dimension will have a distance of n elements. For higher-dimensional data structures, the neighbour property gets even worse, as still only the neighbour relation in one dimension will be retained.
- Continuity of the sequentialisation is violated at the boundaries of the matrix – between elements with successive index, a jump to the new row (or column) will occur, if the respective elements lie at opposite ends of the matrix.
- There is virtually no clustering of data. A given index range corresponds to few successive rows (or columns) of the matrix, and will thus a correspond to a narrow band within the matrix. If the index range is smaller than one column, the band will degenerate into a 1D substructure.
- The row and column dimension are not treated uniformly. For example, many text books on efficient programming will contain hints that loops over 2D arrays should always match the chosen sequentialisation. For a row-major scheme, the innermost loop needs to process the elements of a single row. The respective elements are then accessed successively in memory, which on current processors is executed much faster compared to so-called stride-n accesses, where there are jumps in memory between each execution of the loop body.

The last item, in particular, reveals that choosing row-major or column-major schemes favours the simple indexing over the efficient execution of loops, because how to nest the loops of a given algorithm cannot always be chosen freely.

Row-Major and Column-Major in Numerical Libraries

While up-to-date libraries for problems in numerical linear algebra – both for basic subroutines for matrix-vector operations or matrix multiplication as for more complicated routines, such as eigenvalue computations – are still based on the common row-major or column-major data structures, they invest a lot of effort to circumvent the performance penalties caused by these data structures. Almost always, blocking and tiling techniques are used to optimise the operating blocks to match the size and structure of underlying memory hardware. Often, multiple levels of blocking are necessary.

Nevertheless, due to the simplicity of the row- and column-oriented notations, the respective data structures are kind of hard-wired in our “programming brain”, and we seldom consider alternatives. A simple nested loop, such as

```
for $i = 1, \dots, n$ do
    for $j = 1, \dots, m$ do
        ...
```

will therefore usually not be replaced by a more general construct, such as

```
for all $n$-tupels $(i,j,\dots)$ do
        ...
```

even if the algorithms would allow this. However, to introduce improvements to a data structure in a late stage of program design, is almost always difficult.

In this book, we will discuss sequential orders for data structures using *space-filling curves* – presenting simple problems from linear algebra (in particular, in Chap. 13) as well as examples from other fields in scientific computing, where data structures based on *space-filling curves* can improve the performance of algorithms. Our intention is not to argue that space-filling curves are always the best choice in the end. However, they offer a good starting point to formulate your problems and data structures in a more general way.

↱ What's next?

Remember: you might want to read the chapters of this book in non-sequential order. Boxes like the present one will suggest how to read on.

- ➔ The default choice is usually the next chapter and will be indicated by the forward sign. In the following chapter, we will start with the construction of space-filling curves.
- ↘ This sign indicates that the next chapter(s) might be skipped.
- ↱ The "turn right" sign indicates that you might want to do a slight detour and already have a glimpse at a chapter or a section that comes later in the book. For example, Chap. 9 will deal with quadtrees and octrees in detail – in particular, Sect. 9.1.1 will quantify how many grid cells may be saved by using a quadtree grid.
- ↰ The "turn left" sign will indicate that you might want to redo an earlier section or chapter in the book (perhaps with some new idea in mind).

Chapter 2
How to Construct Space-Filling Curves

In a mathematical sense, introducing a sequential order on a d-dimensional array of elements (or cells) defines a corresponding mapping – from the range of array indices $\{1,\dots,n\}^d$ to sequential indices $\{1,\dots,n^d\}$, and vice versa. From a practical point of view, such sequential orders should result from a *family* of orders, i.e. be generated via a uniform construction for arbitrary n (and maybe d). In scientific computing (and many other fields), the respective data sets will represent continuous data, and the size of the data set will depend on how we chose the resolution in space (or time). We can therefore ask ourselves whether it is possible, or even a good idea, to derive a suitable family of mappings from a *continuous* mapping. In such a setting, our sequential index set becomes a parameter interval, such as the 1D unit interval $[0,1]$. Similarly, the multi-dimensional element set becomes a certain image range, for example the unit interval $[0,1]^d$.

According to our discussion on the efficiency of sequential orders, in Sect. 1.3.1, a good continuous mapping should be *bijective* and *continuous*. A bijective mapping ensures the required one-to-one correspondence between index and indexed element, and a continuous mapping will avoid jumps between the image points of adjacent parameters – which will avoid "holes" in the generated sequential order and improve locality and clustering of data. We will see, in the following section, that these two requirements are already too much for our wish list. But we will introduce the Hilbert and the Peano curve as good alternatives.

2.1 Towards a Bijective Mapping of the Unit Interval to the Unit Square

In set theory, two sets are defined to have the same number of elements, if a bijective mapping exists between the two sets. This is highly intuitive for finite sets, as the bijective mapping will provide a one-to-one matching between all elements, but also works for the cardinality of infinite sets. The *cardinality* of sets, i.e. the quantification of the (possibly infinite) number of elements of sets, is also important

M. Bader, *Space-Filling Curves*, Texts in Computational Science and Engineering 9, DOI 10.1007/978-3-642-31046-1_2, © Springer-Verlag Berlin Heidelberg 2013

in computer science, in particular when talking about infinite sets. Fundamental concepts such as countability and enumerability of sets are derived from cardinality and its definition via the existence of a bijective mapping (from $\mathbb{N}$ to such a set). The existence of functions that are non-computable is one of the important consequences.

Hence, demanding a one-to-one mapping between the unit interval $[0, 1]$ and the unit square $[0, 1]^d$ implies that these need to have the same number of points – which intuitively is not at all to be expected (after all, the unit interval is just a single edge of the unit square). The following, highly contra-intuitive example concerning the cardinality of sets is therefore well-known from many books and lectures in that area, and goes back to Georg Cantor. We consider a mapping between the unit interval $[0, 1]$ and the unit square $[0, 1] \times [0, 1]$. For any argument $t \in [0, 1]$, we can compute its binary representation

$$t = 0_2.b_1b_2b_3b_4b_5b_6\ldots$$

As the image of t, we define the following point in the unit square:

$$f(t) = \begin{pmatrix} 0_2.b_1b_3b_5\ldots \\ 0_2.b_2b_4b_6\ldots \end{pmatrix}.$$

That means, we simply build the first coordinate from all "odd" binary digits, and the second coordinate from all "even" digits. In the same way, we can define an inverse mapping g from the unit square to the unit interval:

$$g\begin{pmatrix} 0_2.b_1b_3b_5\ldots \\ 0_2.b_2b_4b_6\ldots \end{pmatrix} = 0_2.b_1b_2b_3b_4b_5b_6\ldots$$

It is easy to prove that both f and g are *surjective* mappings. Each point of the unit square is an image of a parameter in the unit interval, which we can find via the binary representation. Likewise, each number in the unit interval can be found as a result of g. If the unit square and the unit interval were finite sets, we could conclude that the two sets must have the same number of elements. But is this also true for infinite sets?

It turns out that neither f or g are bijective, though. For example:

$$f\left(\frac{1}{2}\right) = f(0_2.1) = \begin{pmatrix} 0_2.1 \\ 0_2.0 \end{pmatrix} = \begin{pmatrix} \frac{1}{2} \\ 0 \end{pmatrix}$$

and

$$f\left(\frac{1}{6}\right) = f(0_2.0010101\ldots) = \begin{pmatrix} 0_2.01111\ldots \\ 0_2.00000\ldots \end{pmatrix} = \begin{pmatrix} \frac{1}{2} \\ 0 \end{pmatrix}.$$

However, already in 1878 Georg Cantor showed (as corollary of a more general theorem) the existence of a mapping between unit square and unit interval that is indeed bijective. He thus proved that the unit interval has as many points as the unit square – though being a subset of the unit square. This apparent contradiction led to intense discussion about the cardinality of sets. The resulting concepts of countable or enumerable sets are nowadays part of any lecture in computability theory.

2.2 Continuous Mappings and (Space-Filling) Curves

Almost immediately after Cantor's finding, mathematicians began to look for a mapping that is not only bijective, but also continuous. The graph of such mappings is called a *curve*:

Definition 2.1 (Curve). Let $f\colon \mathcal{I} \to \mathbb{R}^n$ be a continuous mapping of the compact set $\mathcal{I} \subset \mathbb{R}$ into $\mathbb{R}^n$. The respective image $f_*(\mathcal{I})$ of such a mapping is then called a *curve*, and $x = f(t), t \in \mathcal{I}$, is called the *parameter representation* of the curve.

Our general understanding of a curve is probably that of a contiguous line, such as an arc, circle, ellipsoid, or similar. The cited mathematical definition is thus much more general: a curve is the image of any continuous mapping from a parameter interval into a 2D or higher dimensional space. The *image* of the mapping is defined as the set of possible values, i.e. $f_*(\mathcal{I}) := \{f(t) \in \mathbb{R}^n \mid t \in \mathcal{I}\}$. Usually, the compact parameter set $\mathcal{I}$ will simply be the unit interval $[0, 1]$. More complicated sets, however, are also possible, as for the Lebesgue curve, for example (see Sect. 7.2).

Hence, finding a bijective curve that maps the unit interval to the unit square means finding a curve that visits each point of the unit square (surjectivity), but does not intersect itself (to remain injective). In 1879, already, Eugen Netto proved that such a curve cannot exist. Such a mapping cannot be bijective and continuous at the same time – provided that the target domain is a square or, in general, a smooth manifold.

In 1890, however, Giuseppe Peano [214] and David Hilbert [131] presented curves that are continuous and *surjective* (but not injective). As these curves still visit every point of the unit square (thus completely filling a certain area or volume), they are called *space-filling*:

Definition 2.2 (Space-filling Curve). Given a curve $f_*(\mathcal{I})$ and the corresponding mapping $f\colon \mathcal{I} \to \mathbb{R}^n$, then $f_*(\mathcal{I})$ is called a *space-filling curve*, if $f_*(\mathcal{I})$ has a Jordan content (area, volume, ...) larger than 0.

Again, this definition is not restricted to the unit square as target domain: if a continuous mapping $f\colon \mathcal{I} \to \mathcal{Q} \subset \mathbb{R}^n$ is also *surjective*, which means that every point of the target set $\mathcal{Q}$ is an image of f, then $f_*(\mathcal{I})$ is a space-filling curve, if the area (or volume) of $\mathcal{Q}$ is non-zero. If $\mathcal{Q}$ is a "well-behaved" domain (i.e. a smooth manifold), then there can be no bijective mapping $f\colon \mathcal{I} \to \mathcal{Q} \subset \mathbb{R}^n$, such that $f_*(\mathcal{I})$ is a space-filling curve (due to the proof by E. Netto).

As f therefore cannot be injective, there are obviously situations where several parameters are mapped to the same image point. Regarding the cardinality of sets, we can therefore claim that the parameter interval has even more points than the target domain $\mathcal{Q}$ (though we already know that we must not rely on such intuition). Regarding the use of such mappings to generate a family of sequential ("space-filling") orders, the resulting ambiguity in indexing could be a problem. However, we will see that the problem is less severe once we move back to discrete sequential orders.

In the following sections, we will discuss the two most prominent examples of *space-filling* curves – the Hilbert curve and the Peano curve. We will discuss their construction, prove that they are indeed space-filling curves (according to the definition given above), and introduce some important terminology.

2.3 The Hilbert Curve

In Sect. 1.1, we discussed a specific sequential order of grid cells in a quadtree that will connect only neighbouring grid cells. The Hilbert curve is the generalisation of this sequentialisation. Its construction and definition as a space-filling curve is based on a recursive procedure, which resembles the quadtree generation:

1. The square target domain, starting with the unit square, is subdivided into four subsquares, each of which with half the side length of the parent square.
2. Find a space-filling curve for each subsquare. The respective curve is obtained as a scaled-down, rotated or reflected version of the original curve.
3. The respective reflection and rotation operations are to be chosen such that the four partial curves can be connected in a way to preserve continuity.

Hence, we need to determine how to perform these reflections and rotations, and also to turn this recursive idea into a formal definition.

2.3.1 Iterations of the Hilbert Curve

The so-called *iterations* of the Hilbert curve are a suitable means to illustrate the reflection and rotation operations and the recursive building principle. The iterations connect, as a polygon, the midpoints of the recursively subdivided subsquares. More precisely, they connect the subsquares of a given recursive level such that all subsquares of this level form a sequence where consecutive subsquares share a common edge. Hence, for each level of refinement, we obtain one iteration of the Hilbert curve. We will therefore talk of the n-th iteration of the Hilbert curve, if that iteration connects the subsquares of the n-th refinement level.

Figure 2.1 shows the first three iterations, and illustrates how these iterations are built step by step. There are actually two ways to explain this construction:

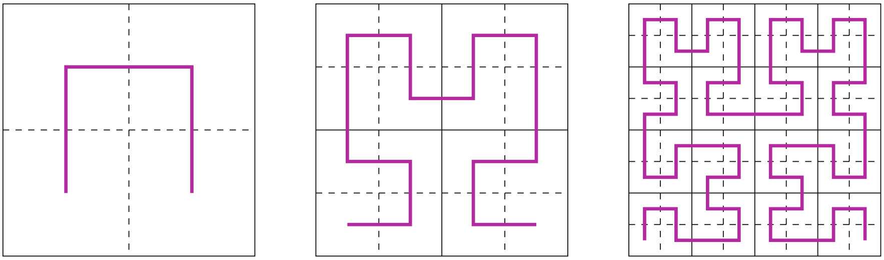


Fig. 2.1 The first three iterations of the Hilbert curve

1. From each iteration to the next, all existing subsquares are subdivided into four smaller subsquares. These four subsquares are connected by a pattern that is obtained by rotation and/or reflection of the fundamental pattern given in the leftmost image of Fig. 2.1. As we know where the iteration curve will enter and exit the existing subsquare, we can determine how to orientate the local pattern.
2. From each iteration to the next, four copies of the existing iteration are connected. The copies are rotated and reflected such that their start and end points can be connected (compare the introduction at the beginning of Sect. 2.3). Note that the orientation of the four iterations is the same regardless of the resolution of the existing iteration.

We will imagine the Hilbert curve as the limit curve that is obtained, if we infinitely repeat this recursive refinement. The iterations themselves, however, will still be of practical importance later on, as we can directly use them for all kinds of finite, discrete data structures, such as grids of cells or matrices.

2.3.2 *Approximating Polygons*

From the iterations of the Hilbert curve, we can anticipate that the "final" Hilbert curve will start in the lower left corner of the unit square – i.e. in point $(0,0)$ –, run through the entire unit square, and terminate in the lower right corner – in point $(1,0)$. If we take a detailed look at the subsquares, and in particular if we refine the iterations accordingly, we see that this also holds for the subsquares: the Hilbert curve will enter each subsquare in one particular corner, and will exit the subsquare in one of the corners that share a common edge with the entry corner.

If we connect these entry and exit corners with a polygon, we obtain the so-called *approximating polygon* of a Hilbert curve. As for the iterations, we obtain different approximating polygons depending on the refinement level of the subsquares. We will speak of the n-th approximating polygon, if it connects the entry and exit corners of the subsquares after n refinement steps. Figure 2.2 shows the first three approximating polygons of the Hilbert curve.

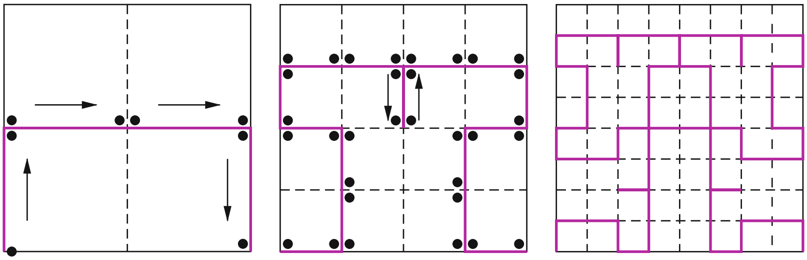


Fig. 2.2 The first three approximating polygons of the Hilbert curve; entry and exit points, as well as the sequence in which they are visited by the Hilbert curve, are marked in addition

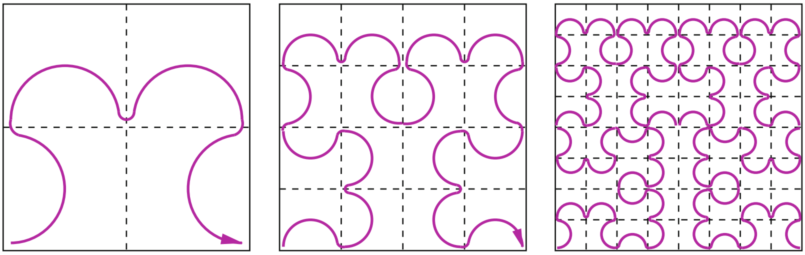


Fig. 2.3 Illustration of the Hilbert curve mixing properties of iterations and approximating polygons

The approximating polygons are highly useful during the construction of the Hilbert curve. Together with the iterations, they determine the sequence in which the subsquares are visited by the Hilbert curve. In Chap. 5, the approximating polygons will be discussed in more detail.

The concept of iterations and approximating polygons is often mixed to generate a representation that illustrates both the entry and exit points as well as the sequential order generated on the subsquares. Figure 2.3 gives an example of such a representation.

2.3.3 Definition of the Hilbert Curve

For a mathematically strict definition of the Hilbert curve, we will use sequences of nested 1D and 2D intervals. The fundamental idea of this construction is that the Hilbert curve shall run through the unit square with a constant velocity: after it has covered one quarter of the entire square, we request that it has also covered one quarter of the parameter interval – and this shall also hold during the recursive refinement of squares and parameter intervals.

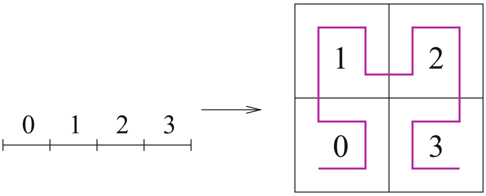


Fig. 2.4 Nested intervals and mapping parameter intervals to 2D subsquares during the construction of the Hilbert curve

Hence, for a given parameter t, we construct a sequence of nested intervals as follows:

- As starting interval, we chose the entire parameter interval [0, 1].
- For each interval $[a, a+h]$, its successor it chosen as one of the intervals

$$\left[a, a+\frac{h}{4}\right], \left[a+\frac{h}{4}, a+\frac{2h}{4}\right], \left[a+\frac{2h}{4}, a+\frac{3h}{4}\right], \left[a+\frac{3h}{4}, a+h\right], \tag{2.1}$$

  i.e. the first, second, third, or fourth quarter of $[a, a+h]$.
- Of course, the successor is chosen such that it contains t.

For example, for the parameter $t = \frac{1}{3}$, we will obtain the following sequence of intervals:

$$[0,1], \left[\frac{1}{4}, \frac{2}{4}\right], \left[\frac{5}{16}, \frac{6}{16}\right], \left[\frac{21}{64}, \frac{22}{64}\right], \ldots$$

According to our recursive idea, each interval of this sequence is mapped to a specific subsquare. Analogous to the sequence of nested intervals, we construct a sequence of nested subsquares. Again, each such subsquare $[a, b] \times [c, d]$ will be one quarter of the parent square:

$$\begin{gathered}\left[a, \frac{a+b}{2}\right] \times \left[c, \frac{c+d}{2}\right], \quad \left[\frac{a+b}{2}, b\right] \times \left[c, \frac{c+d}{2}\right], \\ \left[a, \frac{a+b}{2}\right] \times \left[\frac{c+d}{2}, d\right], \quad \left[\frac{a+b}{2}, b\right] \times \left[\frac{c+d}{2}, d\right].\end{gathered} \tag{2.2}$$

Which of these 2D intervals is to be chosen can be read from the iterations or also from the approximating polygons of the Hilbert curve – as illustrated in Fig. 2.4.

Note that if t lies on an interval boundary, the choice of nested parameter intervals (and, as a consequence, that of the subsquares) is not unique. Whether this will pose problems for our definition of the Hilbert curve, will need to be discussed later.

Definition 2.3 (Hilbert Curve). Let the parameter representation $h(t)$ be defined via the following mapping algorithm:

- For each parameter $t \in \mathcal{I} := [0, 1]$, a sequence of nested intervals

$$\mathcal{I} \supset [a_1, b_1] \supset \ldots \supset [a_n, b_n] \supset \ldots,$$

  exists, such that each interval is obtained by splitting its predecessor into four parts of equal size – as in Eq. (2.1).

- Any such sequence of intervals can be mapped one by one to a sequence of nested 2D intervals, as defined in Eq. (2.2). The iterations of the Hilbert curve determine which 2D interval (i.e. which subsquare) is the image of which interval.
- The constructed 2D nested intervals will converge to a uniquely defined point in $\mathcal{Q} := [0, 1] \times [0, 1]$ – this point shall be $h(t)$.

The image of $h : \mathcal{I} \to \mathcal{Q}$ is then a space-filling curve, the *Hilbert curve*.

2.3.4 Proof: h Defines a Space-Filling Curve

To prove that the defined mapping h really determines a space-filling curve, we need to prove the following three properties of h:

1. h is a mapping, i.e. each $t \in \mathcal{I}$ is mapped to a *unique* value $h(t)$.
2. $h: \mathcal{I} \to \mathcal{Q}$ is *surjective*.
3. h is *continuous*.

Proof: *h* Defines a Mapping

The question whether h defines a mapping is a direct consequence of the fact that the nested intervals used in the definition are not uniquely defined. If a given parameter t is identical to one of the interval boundaries, there are two possible choices to proceed with the nested intervals. We therefore need to prove that the value $h(t)$ is independent of the choice of nested intervals. To prove this property on the basis of our current definition would be rather laborious. Hence, we will come back to this question in Sect. 4.3, when we will have better tools available.

Until then, we can use the continuity of h as justification: the two different choices of nested intervals are equivalent to computing both one-sided limits of $h(t)$ at the parameter t. Continuity of h will assure that the limit from the left and the limit from the right are equal.

Proof: *h* Is Surjective

To prove surjectivity, we need to prove that for each point q of the unit square, there is a parameter t, such that $h(t) = q$. That proof can be obtained by going backwards from the nested subsquares to the nested intervals:

- For each point $q \in \mathcal{Q}$, we can construct a nested sequence of 2D intervals, such that these 2D intervals correspond to the subsquares occurring in the construction of the Hilbert curve.
- By construction, this 2D nesting uniquely corresponds to a sequence of nested intervals in the parameter interval $\mathcal{I}$. Due to the completeness of the real numbers,

and because $\mathcal{I}$ was assumed to be compact, the nested intervals will converge to a parameter t.

- With these nested parameter intervals, and the corresponding 2D nested intervals, we apply the definition of the Hilbert curve and directly obtain that $h(t) = q$. Hence, h is surjective.

At this point, let us again have a look at the uniqueness problem: In the definition of the Hilbert curve, we could have used a unique definition, for example by using the following (right-open) intervals:

$$\left[a, a + \frac{h}{4}\right), \left[a + \frac{h}{4}, a + \frac{2h}{4}\right), \left[a + \frac{2h}{4}, a + \frac{3h}{4}\right), \left[a + \frac{3h}{4}, a + h\right).$$

However, such a nested sequence of intervals might actually converge to a limit point that is not contained in any of the intervals. For example, if we always choose the rightmost interval in such a sequence, the limit will be the right (open!) boundary point, which is not included in the interval. Hence, when using intervals with open boundaries, we would run into even more problems compared to using compact intervals. And the uniqueness problem will turn out to be not that nasty.

2.3.5 Continuity of the Hilbert Curve

We have shown that h is a surjective mapping from the unit interval to the unit square – i.e. h is a *space-filling* mapping. To proof that h is a curve, as well, we need to show that h is *continuous*. We will actually prove a slightly stronger property, as h turns out to be even *uniformly* continuous.

A function $f\colon \mathcal{I} \to \mathbb{R}^n$ is *uniformly continuous* on an interval $\mathcal{I}$, if for each $\epsilon > 0$ there exists $\delta > 0$, such that for any $t_1, t_2 \in \mathcal{I}$ with $|t_1 - t_2| < \delta$, we have that $\|f(t_1) - f(t_2)\|_2 < \epsilon$. Remember that for the "regular" continuity, it would be sufficient to show continuity for each parameter t in the interval $\mathcal{I}$ separately. Hence, we would be allowed to choose δ depending on t, as well.

Proving Continuity

Our strategy for the proof is to find an estimate that will give us an upper bound for the distance of the image points, $\|h(t_1) - h(t_2)\|_2$, depending on the distance of the parameters, $|t_1 - t_2|$. If we can obtain such an estimate for an arbitrary choice of the parameters t_1 and t_2, we can easily find a proper δ for any ϵ that is given to us. Such an estimate is obtained in four main steps:

1. Given two parameters $t_1, t_2 \in \mathcal{I}$, we choose a refinement level $n \in \mathbb{N}$, such that $|t_1 - t_2| < 4^{-n}$.

2. For this n-th refinement level, the length of the nested intervals of the Hilbert curve construction is 4^{-n}. Therefore, the interval $[t_1, t_2]$ can range over at most two such nested intervals, which have to be neighbours – because, if an entire interval could fit between t_1 and t_2, their distance would have to be larger than 4^{-n}.
3. Due to the construction principle of the Hilbert curve, any two neighbouring intervals will be mapped to two subsquares that share a common edge. By construction, these squares will have a side length of 2^{-n}. The image points $h(t_1)$ and $h(t_2)$ both have to be within the rectangle that is built from these two subsquares.
4. The distance between $h(t_1)$ and $h(t_2)$ can therefore not be larger than the length of the diagonal of this rectangle. The rectangle's side lengths are 2^{-n} and $2 \cdot 2^{-n}$, hence the length of the diagonal is $2^{-n} \cdot \sqrt{5}$ (rule of Pythagoras). This length is therefore an upper bound for the distance of $h(t_1)$ and $h(t_2)$, and we obtain: $\|h(t_1) - h(t_2)\|_2 \leq 2^{-n}\sqrt{5}$.

For a given $\epsilon > 0$, we can now always pick an n, such that $2^{-n}\sqrt{5} < \epsilon$. With this n, we obtain a $\delta := 4^{-n}$, such that for any t_1, t_2 with $|t_1 - t_2| < \delta$, we obtain the continuity requirement from the considerations above:

$$\|h(t_1) - h(t_2)\|_2 \leq 2^{-n}\sqrt{5} < \epsilon.$$

This proves continuity of h.

At this point, we should note that we only required two major properties of the construction to prove continuity:

- h always maps two adjacent parameter intervals to two adjacent subsquares (that share a common edge). Note that, so far, we claimed that the Hilbert curve meets this condition "by construction", but didn't give a real proof. We will be able to provide such as proof in Sect. 3.4.
- Both intervals and subsquares are recursively subdivided into quarters. This will ensure the exponential decay of interval and side lengths. It will also make sure that all intervals and subsquares of the same refinement level have the same size. (Note that we also use the fact that the squares stay squares, i.e. preserve their shape during refinement.)

As many other space-filling curves also rely on these (or similar) properties, we will be able to re-use our proof of continuity for a lot of further space-filling curves.

2.3.6 Moore's Version of the Hilbert Curve

As illustrated in Fig. 2.5, it is possible to combine four regular Hilbert iterations (or Hilbert curves) and obtain a *closed* space-filling curve, i.e., one that has the same start and end point – here in $\left(0, \frac{1}{2}\right)$. We will call the respective space-filling curve *Hilbert-Moore* curve, as it was found by E. H. Moore.

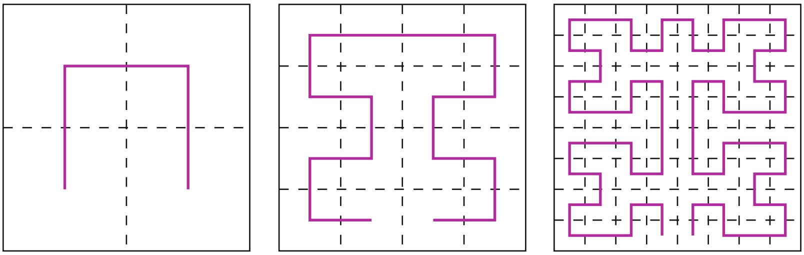


Fig. 2.5 The first three iterations of Moore's version of the Hilbert curve

For construction and description of the Hilbert-Moore curve, see the exercises after Chaps. 3 and 4.

2.4 Peano Curve

Following the same construction as for the Hilbert curve, we can also define the historically first space-filling curve – the *Peano* curve, found by G. Peano in 1890. Again, we follow a recursive construction of iterations of the Peano curve, and define the Peano curve via the limit of this construction. The construction of the Peano curve works as follows:

1. Whereas the Hilbert curve is based on a recursive substructuring into four subsquares in each step, the Peano curve construction subdivides each subsquare into *nine* congruent subsquares, each with a side length equal to one third of the previous one.
2. Correspondingly, we define the nested intervals via a recursive subdivision into nine equal-sized subintervals in each step. Again, each interval is mapped to a specific subsquare of the same recursion level.
3. Hence, each of the subsquares will contain a part of the final Peano curve that is again a suitably scaled and transformed copy of the entire curve. As illustrated in Fig. 2.6, we need to assemble these partial curves to a contiguous curve on the unit square.
4. The required transforms can also be determined from Fig. 2.6. In contrast to the Hilbert curve, no rotations are required – we only require horizontal or vertical reflections. Note that the reflection at the horizontal axis are especially required to obtain the correct orientation of the curve in the central (top-centre, mid-centre, and bottom-centre) subsquares.
5. The resulting switch-back patterns (here, in the vertical variant) are characteristic for the Peano curve, and occur on every level of the Peano iterations.

The dominating direction of the "switch-backs" can be both horizontal and vertical – see the two examples plotted in Fig. 2.7.

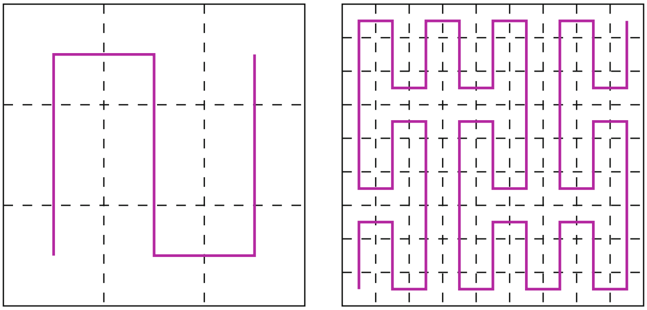


Fig. 2.6 The first two iterations of the Peano curve

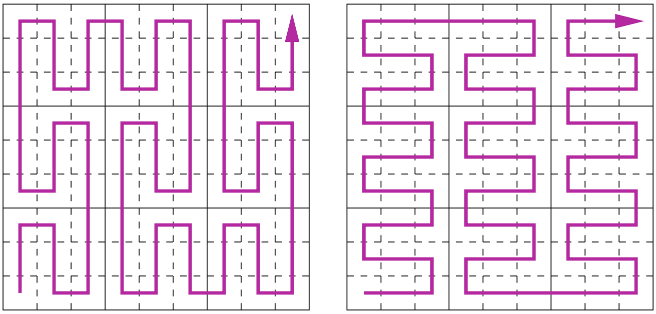


Fig. 2.7 Two different Peano curves of the switch-back type

The switch-back can also be in alternate directions, as in the example shown in Fig. 2.8a. In that way, a multitude of different Peano curves can be generated. In principle, there are 2^9 possible variants, however, a lot of them are symmetric to each other. According to Sagan [233], there are 272 unique variants of switch-back-type Peano curves. In addition, there are two variants of the so-called *Peano-Meander* curves – the first iteration of one of these curves is given in Fig. 2.8b.

In contrast, there is indeed only one 2D Hilbert curve, if we disregard symmetric variants obtained via reflection or rotation. A further variant of the Hilbert curve would be the Hilbert-Moore curve, however, this variant has different start and end points of the curve.

Continuity and Surjectivity of the Peano Curve

To prove that the Peano construction indeed leads to a space-filling curve, we again have to prove continuity and surjectivity of the respective mapping. Luckily, we can re-use the proof for the Hilbert curve almost word-by-word:

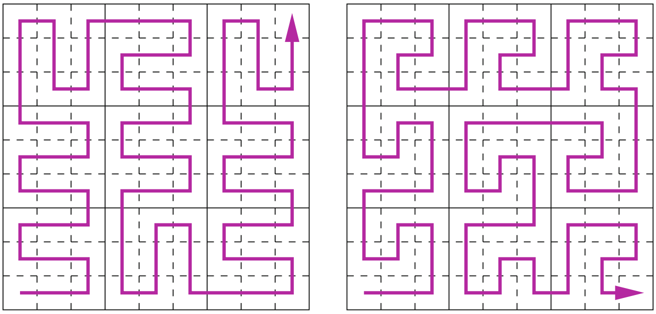

(a) Switch-back Peano curve (b) Peano-Meander curve

Fig. 2.8 Peano curves of the switch-back vs. the Meander type. The switch-back variant (**a**) has been introduced as "Meurthe order" [123]

- *Surjectivity* results from the synchronous nesting of intervals and subsquares: for each point of the unit square, there is a suitable sequence of nested subsquares The corresponding intervals form a sequence of nested intervals that converge to the parameter mapped to the given point. Obviously, that also includes existence of such a point.
- To prove continuity, we remember that the continuity of the Hilbert curve was proven based on two substantial properties:

  1. Two adjacent intervals are always mapped to adjacent subsquares, and
  2. Intervals and subsquares are generated via a recursive substructuring process that splits each subsquare into a set of scaled-down, equal-sized subsquares (analogous for the intervals).

  Both requirements are satisfied for the Peano construction. We can therefore infer continuity from the respective proof for the Hilbert curve.

2.5 Space-Filling Curves: Required Algorithms

Applications of space-filling curves basically exploit the particular order space-filling curves induce on two- and higher-dimensional data structures. Our first step in that direction was to demonstrate that space-filling curves exist, and to provide a rough description of how to construct them. However, to be able to exploit space-filling curves for computational problems, we need more formal descriptions of space-filling curves – and of the induced family of space-filling orders for indexing – and turn these into useful algorithms.

But what operations are we likely to need to be able to work with space-filling curves? First of all, we will require functions that compute the mapping between a higher-dimensional data structure and its sequentialisation by a space-filling curve:

Computation of function values:
For a given parameter $t \in \mathcal{I}$, we need to be able to compute the corresponding point $h(t)$ on the space-filling curve, i.e. we require an algorithm to compute the respective *mapping* h. For the example of data indexed by a space-filling curve, we need to be able to recover the data, if the index is given.

Computation of the index of a point:
For a given point $p \in \mathcal{Q}$, we want to compute the parameter t, for which $h(t) = p$. A main problem, here, will be that the inverse mapping h^{-1} is not well defined, as several parameters may be mapped to the same point (i.e., the function h is not bijective). We will see, however, that we can at least compute an inverse mapping $\bar{h}^{-1}$ that provides such a parameter in a unique way.

A further typical operation will be to simply process all data in a certain order:

Traversal of objects indexed by *h*:
Given a set of points $p_i \in \mathcal{Q}$, we want to process the points in the order induced by the space-filling curve, i.e. such that

$$\bar{h}^{-1}(p_{i_0}) < \bar{h}^{-1}(p_{i_1}) < \dots$$

The data points may be regularly (for multi-dimensional arrays, e.g.) or irregularly (geographical coordinates, e.g.) distributed in space.

The latter task is similar to a computer science "classic" – the *travelling salesman* problem. There, the task is to visit a given number of cities (given by their geographical position), such that the entire travelled distance is minimised. Space-filling curves can actually provide a pretty good heuristics to solve this problem (see Chap. 15 for some further details). An example for a traversal of regularly distributed array data would be, for example, an update of all pixels in a computer image that is stored as an array.

The computation of function values of indices, as well as the implementation of traversals along a space-filling curve require a better, mathematical or computational description of the curves. The definition used so far is not exact enough to be transferable into a computer program. We will first discuss grammar-based descriptions of space-filling curves, in Chap. 3, which will lead to traversal algorithms. In Chap. 4, we will present an arithmetic representation of space-filling curves that allows an efficient computation of mappings and indices.

References and Further Readings

Sagan's book on space-filling curves [233] not only provides a thorough introduction to the construction and mathematical analysis of space-filling curves, but also gives extensive information on the history of space-filling curves (see also his article [230]). Hence, we can keep this references section comparably short.

Note that the original papers by Peano [214] (in French) and Hilbert [131] (in German) are actually available online via the Digitalisierungszentrum Göppingen. Hilbert introduced the geometrical approach to construct space-filling curves via the respective iterations. Peano's description was based on the ternary representation of numbers – see Sect. 8.2.4. Moore introduced his variant of Hilbert's curve in 1900 [193] (four additional patterns for curves based on the Hilbert curve were given by X. Liu [168]). In addition, Moore gave a formal proof that the Hilbert and Peano curves are continuous but do not have a derivative in any point of the curve. He also discussed in detail the construction of the curves via "broken-line curves" – which we refer to as the iterations and approximating polygons of the space-filling curves.

Wunderlich [277], in 1973, gave an instructive representation of the Hilbert and Peano curves (also the Sierpinski curve introduced in Chap. 6), where he also discusses the geometrical construction of space-filling curves via recursive use of basic patterns (in German: *"Grundmotive"*). For the Peano curves, in particular, he discussed the switch-back (*Serpentinentyp*) versus Meander type and introduced a simple notation for the different variants obtained for the switch-back type. In his notation, the standard curve in Fig. 2.6 is encoded by the bit-9-tuple 000 000 000, while the right curve in Fig. 2.7, where all nine sub-patterns are switched to horizontal direction would be Serpentine 111 111 111. Consequently, the curve in Fig. 2.8a is Serpentine 110 110 110. Haverkort and van Walderveen [123] found that the latter curve has the best locality properties (according to certain measures – see Chap. 11) among all switch-back curves, and named it Meurthe order (after the respective river).

What's next?

- The next two chapters will take on the challenge of finding algorithms for indexing and traversal of data structures using space-filling orders. For that purpose, we will need proper (mathematical) descriptions of space-filling curves.
- If you are interested in how a Hilbert or Peano curve might look in 3D, then you can have a quick look at Chap. 8.

Exercises

2.1. Based on Definition 2.3, try to find the parameters (i.e., respective nested intervals for the parameters) that are mapped to the corner points of the approximating polygon of the Hilbert curve.

2.2. The Hilbert curve is determined as soon as the start and end point of the curve are fixed: if start and end are located at corners located on a common edge of the unit square, the rotated and/or reflected versions of the original Hilbert curve are obtained. For the Hilbert-Moore curve, start and end are located on an edge's centerpoint. Are there further possibilities? Try to find all of these (see also [168]).

2.3. Sketch the full proof of the continuity of the Peano curve, and especially compute an upper bound of the distance of points, $\|p(t_1) - p(t_2)\|$ depending on the distance of the parameters, $|t_1 - t_2|$.

2.4. Draw the first iterations of all 272 different Peano curves.

Chapter 3
Grammar-Based Description of Space-Filling Curves

3.1 Description of the Hilbert Curve Using Grammars

To construct the iterations of the Hilbert curve, we recursively subdivided squares into subsquares, and sequentialised the respective subsquares by the recursive patterns given by the iterations. The patterns were obtained by respective rotations and reflections of the original pattern. We will now figure out how many different patterns actually occur in these iterations. For that purpose we will identify the *basic patterns* within the iterations – each basic pattern being a section of the iteration that either corresponds to a scaled down 0-th iteration (sequence "up–right–down") or results from a rotation or reflection of this pattern. Figure 3.1 illustrates the patterns that occur in the first three iterations of the Hilbert curve.

We observe that the iterations consist of only four basic patterns – marked by the letters H, A, B, and C in Fig. 3.1. We also see that, from each iteration to the next, the corresponding part of the iteration will be refined according to a fixed replacement scheme. This scheme is illustrated in Fig. 3.2. The replacement scheme is obviously a closed scheme, such that no additional patterns can occur. In the following, we will therefore describe the successive generation of iterations of space-filling curves by specific *grammars*.

Defining a Grammar to Describe the Iterations of the Hilbert Curve

We will define the respective grammar by the following components:

1. A set of **non-terminal symbols**: $\{H, A, B, C\}$. The non-terminals represent our four basic patterns. The non-terminal H (i.e. the basic pattern) shall be distinguished as the **start symbol**.
2. A set of **terminal symbols**: $\{\uparrow, \downarrow, \leftarrow, \rightarrow\}$. The terminal symbols describe the transitions between the basic patterns, and will also determine the connections

M. Bader, *Space-Filling Curves*, Texts in Computational Science and Engineering 9, DOI 10.1007/978-3-642-31046-1_3, © Springer-Verlag Berlin Heidelberg 2013

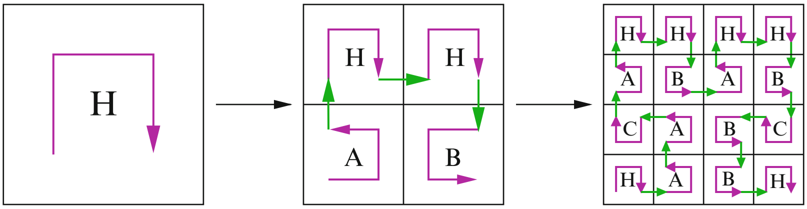


Fig. 3.1 Determining the basic patterns in the first three iterations of the Hilbert curve

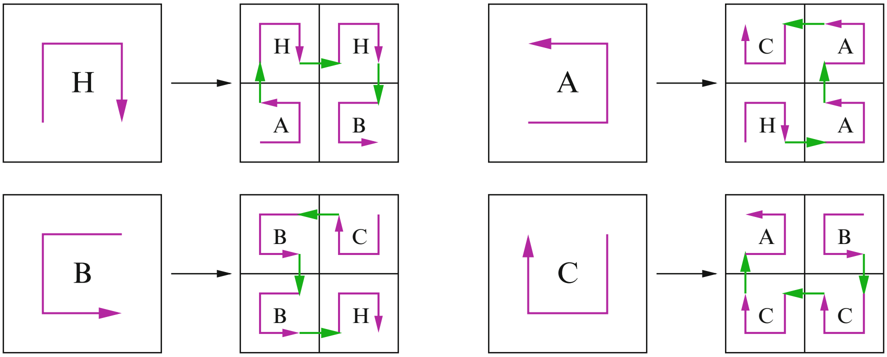


Fig. 3.2 Replacement scheme of the basic patterns in the iterations of the Hilbert curve

between the subsquares by an iteration of the Hilbert curve. In Fig. 3.2, the terminals are represented by the green arrows.

3. A set of **production rules**, i.e. replacement rules for the patterns:

$$H \longleftarrow A \uparrow H \rightarrow H \downarrow B$$

$$A \longleftarrow H \rightarrow A \uparrow A \leftarrow C$$

$$B \longleftarrow C \leftarrow B \downarrow B \rightarrow H$$

$$C \longleftarrow B \downarrow C \leftarrow C \uparrow A$$

   The productions determine for each basic pattern how it has to be replaced during refinement of the iteration – in particular, it determines the correct sequences of basic patterns and transfers between them.

4. An **additional derivation rule**: to construct a string from our grammar, we are only allowed to replace **all non-terminal symbols at once** in a derivation step. We thus ensure that a uniform refinement level is retained throughout the construction of the corresponding curve.

The production rules of our grammar satisfy the requirements for a context-free grammar, as they are frequently used in computer science. However, we did not include any terminal productions, i.e. productions that only have terminal symbols in the right-hand side. Also, the additional derivation rule leads to a modified grammar, which is related to so-called **L-systems** (see the references section).

The Grammar as a Closed System of Patterns

As a side result, the grammar for the Hilbert curve gives an answer to a question we have not yet asked, but actually should have: is the system of patterns and partial iterations to construct the Hilbert curve complete, such that we can infinitely repeat it? Our grammar clearly answers that question, as we can see that only four basic patterns occur in the iterations – represented by the four non-terminals H, A, B, and C. In addition, we see from the production rules, and from the respective illustration in Fig. 3.2 that the continuous connection between the basic patterns is always guaranteed.

Last, but not least, the grammar serves as a mathematically exact description of the sequence of transforms and pattern orientations, which complements the definition of the Hilbert curve. So far, we were only able to give an intuitive, informal description that was based on drawing the first couple of iterations.

Finally, we can interpret the non-terminals H, A, B, and C as representatives of scaled-down "infinite" Hilbert curves, where H represents a Hilbert curve in its original orientation, whereas A, B, and C correspond to the Hilbert curve in different orientations. In that infinite case, the productions then become similar to fixpoint equations.

Strings Generated by the Grammar

We will now discuss how the strings generated by the grammar look, and how they can be used. For that purpose, we will apply the production rules, following the additional derivation rule, and obtain:

$$\begin{aligned} H &\longleftarrow A \uparrow H \rightarrow H \downarrow B \\ &\longleftarrow H \rightarrow A \uparrow A \leftarrow C \uparrow A \uparrow H \rightarrow H \downarrow B \rightarrow A \uparrow H \rightarrow H \downarrow B \downarrow C \leftarrow B \downarrow B \rightarrow H \end{aligned}$$

Note that the arrow symbols exactly describe the iterations of the Hilbert curve, if we interpret them as plotting instructions. The grammar strings provide the sequence of plotting commands, such as "move up/down" or "move left/right" to successively draw an iteration. The scheme is also similar to the concept of *turtle graphics*, as it was used in classic programming languages. There, a "turtle" could be moved over the screen subject to simple commands, such as "move forward", "turn left", or "turn right". We will also discuss grammars to generate the marching instructions for such a turtle, in Sect. 3.4.

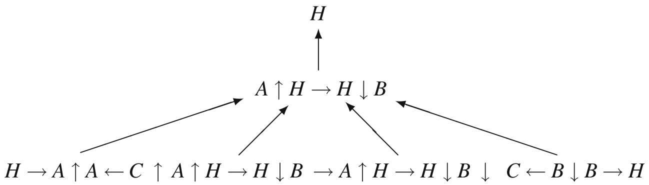

H
A ↑ H → H ↓ B
H → A ↑ A ← C ↑ A ↑ H → H ↓ B → A ↑ H → H ↓ B ↓ C ← B ↓ B → H

Fig. 3.3 The first two levels of the derivation tree for the Hilbert grammar

3.2 A Traversal Algorithm for 2D Data

The iterations of the Hilbert curve give us a recipe to traverse the subsquares that are used to define the Hilbert curve. For two-dimensional data, we can use the generated sequential order as a traversal algorithm. Such data could, for example, be:

- Image data, where images are rasterised using square or rectangular pixels;
- Matrices in linear algebra;
- In general, all data that can be stored as a two-dimensional array.

We obtain a respective traversal algorithm, if we successively generate the strings generated by the grammar, and then execute the moves given by the terminal arrows. This derivation of the strings can be illustrated via a derivation tree, as given in Fig. 3.3. The derivation tree is obviously directly equivalent to the call tree of respective recursive procedures. We will therefore identify the non-terminals H, A, B, and C with corresponding recursive procedures (or methods), and turn the terminal symbols $\uparrow$, $\downarrow$, $\leftarrow$, and $\rightarrow$ into non-recursive procedures that implement the steps in upward, downward, left, or right direction on the respective data structure.

The non-terminal H, as well as the respective production $H \longleftarrow A \uparrow H \rightarrow H \downarrow B$, is thus translated into a recursive procedure `H`, as given in Algorithm 3.1. The procedures `A()`, `B()`, and `C()` are implemented in the same way from the respective non-terminals and productions. The procedures `up()`, `down()`, etc. need to implement the elementary moves on the traversed data structures.

Introducing Terminal Productions

In Algorithm 3.1, the procedures `H()`, `A()`, `B()`, and `C()` do not perform any action; in particular, the case `depth = 0` is disregarded. This corresponds to the fact that we only consider the terminal arrow symbols in the strings of the grammar, and neglect the non-terminals. We could reflect this by introducing ε-productions into the grammar:

Algorithm 3.1: Hilbert traversal (depth prescribed as parameter)

```
Procedure H(depth) begin
    if depth > 0 then
        A(depth-1); up();
        H(depth-1); right();
        H(depth-1); down();
        B(depth-1);
    end
end
Procedure A(depth) begin
    if depth > 0 then
        H(depth-1); right();
        A(depth-1); up();
        A(depth-1); left();
        C(depth-1);
    end
end
Procedure B(depth) begin
    if depth > 0 then
        C(depth-1); left();
        B(depth-1); down();
        B(depth-1); right();
        H(depth-1);
    end
end
Procedure C(depth) begin
    if depth > 0 then
        B(depth-1); down();
        C(depth-1); left();
        C(depth-1); up();
        A(depth-1);
    end
end
```

$$\begin{array}{lcl}
H \longleftarrow A \uparrow H \rightarrow H \downarrow B & | & H \longleftarrow \varepsilon \\
A \longleftarrow H \rightarrow A \uparrow A \leftarrow C & | & A \longleftarrow \varepsilon \\
B \longleftarrow C \leftarrow B \downarrow B \rightarrow H & | & B \longleftarrow \varepsilon \\
C \longleftarrow B \downarrow C \leftarrow C \uparrow A & | & C \longleftarrow \varepsilon
\end{array}$$

The derivation rule then has to be extended: in each derivation step, we have to replace all non-terminals at once, and we have to uniformly apply either the first productions or the ε-productions, everywhere.

As an alternative to the ε-productions, we could also include productions that represent an action to be performed at the current position. We could, for example, use a $\bullet$ as additional terminal symbol:

Algorithm 3.2: Hilbert traversal (execute task)

```
Procedure H(depth) begin
    if depth = 0 then
        // Execute task on current position
        execute(...);
    else
        A(depth-1); up();
        H(depth-1); right();
        H(depth-1); down();
        B(depth-1);
    end
end
```

$$
\begin{array}{lcl}
H \longleftarrow A \uparrow H \rightarrow H \downarrow B & | & H \longleftarrow \bullet \\
A \longleftarrow H \rightarrow A \uparrow A \leftarrow C & | & A \longleftarrow \bullet \\
B \longleftarrow C \leftarrow B \downarrow B \rightarrow H & | & B \longleftarrow \bullet \\
C \longleftarrow B \downarrow C \leftarrow C \uparrow A & | & C \longleftarrow \bullet
\end{array}
$$

The respective grammar translates into Algorithm 3.2, which performs a traversal of some 2D data structure following a Hilbert iteration, and performs a certain task on each element. Try to use this template algorithm to implement a specific task – such as a matrix update or matrix-vector multiplication, as suggested in Exercise 3.3.

Computational Costs of the Traversal Algorithm

We should do a quick calculation of the number of function calls executed by Algorithm 3.2. Assume that, at start, the parameter `depth` contains the value p. We will then generate a Hilbert iteration that traverses the cells of a $2^p \times 2^p$ grid. From each recursion level to the next, the number of recursive calls to one of the procedures `H()`, `A()`, `B()`, or `C()` grows by a factor of 4. Hence, the total number of calls to these procedures is

$$1 + 4 + 4^2 + \ldots 4^p = \frac{1 - 4^{p+1}}{1 - 4} \leq \frac{4}{3} 4^p. \tag{3.1}$$

In addition, on the leaf level each of the 4^p recursive calls issues one terminal call to `execute()`.

As $4^p = 2^{2p} = (2^p)^2$, we see that the number of calls to `execute()` is exactly equal to the number $N = 2^p \times 2^p$ of cells (as it should be). However, we also obtain that the total number of recursive calls is $\mathcal{O}(N)$, due to Eq. (3.1). Note that the number of calls to the recursive functions can be reduced from $\frac{4}{3}N$ to $\frac{1}{3}N$, if we do recursion unrolling – as in Algorithm 3.3. There, the 4^p calls on the leaf level are saved.

Algorithm 3.3: Hilbert traversal with recursion unrolling

```
Procedure H(depth) begin
    if depth = 1 then
        execute(...);up();
        execute(...);right();
        execute(...);down();
        execute(...);
    else
        A(depth-1);up();
        H(depth-1);right();
        H(depth-1);down();
        B(depth-1);
    end
end
```

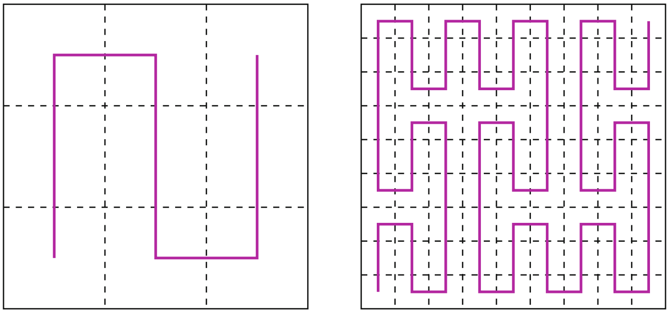


Fig. 3.4 The first two iterations of the Peano curve

3.3 Grammar-Based Description of the Peano Curve

In the same way as for the Hilbert curve, we can derive a grammar for the Peano curve given in Fig. 3.4. The relation between grammar and recursive construction of the Peano curve is illustrated in Fig. 3.5. Note that the patterns P and R, as well as Q and S, only differ in the orientation of the curve.

The grammar is built analogously to the Hilbert curve:

- As non-terminals, we define the characters $\{P, Q, R, S\}$, where the characters again represent the basic patterns, as given in Fig. 3.5. P is defined as the start symbol of the grammar.
- The set of terminal symbols is identical to that of the Hilbert curve: $\{\uparrow, \downarrow, \leftarrow, \rightarrow\}$.

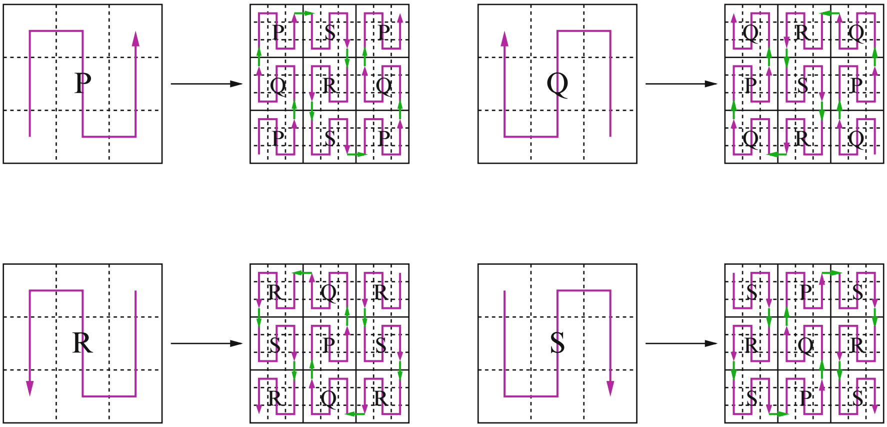


Fig. 3.5 Construction of the Peano curve and the corresponding grammar

Algorithm 3.4: 2D Peano traversal

```
Procedure P(depth) begin
    if depth > 0 then
        P(depth-1); up();
        Q(depth-1); up();
        P(depth-1); right();
        S(depth-1); down();
        R(depth-1); down();
        S(depth-1); right();
        P(depth-1); up();
        Q(depth-1); up();
        P(depth-1);
    end
end
```

- Finally, we read the production rules from Fig. 3.5:

$$P \longleftarrow P \uparrow Q \uparrow P \rightarrow S \downarrow R \downarrow S \rightarrow P \uparrow Q \uparrow P$$

$$Q \longleftarrow Q \uparrow P \uparrow Q \leftarrow R \downarrow S \downarrow R \leftarrow Q \uparrow P \uparrow Q$$

$$R \longleftarrow R \downarrow S \downarrow R \leftarrow Q \uparrow P \uparrow Q \leftarrow R \downarrow S \downarrow R$$

$$S \longleftarrow S \downarrow R \downarrow S \rightarrow P \uparrow Q \uparrow P \rightarrow S \downarrow R \downarrow S$$

Again we demand, as an additional derivation rule, that in strings generated by the grammar, we are only allowed to expand all non-terminals *at once*.

Again , we can turn this grammar into a corresponding algorithms that performs a 2D-traversal following the iterations of a Peano curve. The procedure `P()`, resulting from symbol P, is given in Algorithm 3.4.

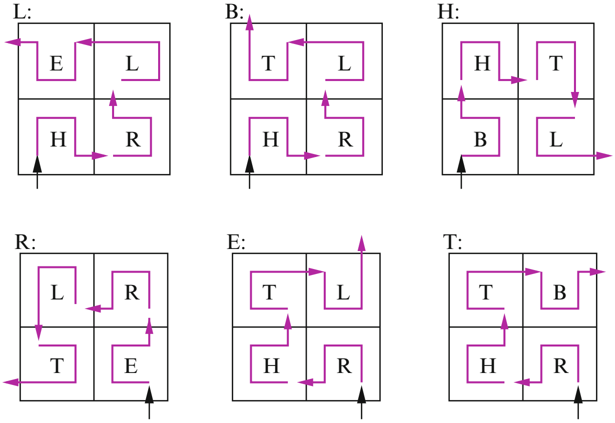


Fig. 3.6 Construction of the turtle grammar for the Hilbert curve

3.4 A Grammar for Turtle Graphics

In the previous sections, we have developed traversal algorithms that can traverse a 2D data structure along a space-filling curve. We used a plotter concept, where the plotter's pen obeys commands such as *up*, *down*, *left*, and *right*, which means it is navigated using absolute directions. While this is sufficient for a traversal algorithm, it also poses certain limits. Consider, for example, Algorithm 3.2, which adds the execution of a task to every visit of a data element. At the point where this task execution is called by the recursive function, it can no longer tell which element was previously visited, and it can also not predict to which element it will move next. From the current pattern, it would be able to determine a recursive refinement of the data structure, but for both entry and exit, there are two possible directions. These are given by the terminal calls to `up()`, `down()`, etc., but the previous call is no longer accessible on the call stack, and the following call is still to be determined by a parent cell.

In the following, we will derive a grammar that helps to determine the entry and exit directions. It will also change the plotter concept into that of turtle graphics, where we navigate using relative directions – i.e. the turtle obeys to commands such as *forward*, *turn left*, or *turn right*. As a result, the orientation of the patterns no longer matters: for the Hilbert pattern H, the turtle does the same relative moves (*forward, turn right, forward, turn right, forward*) as for pattern C. We are therefore left with only two patterns, but each of these patterns splits up into several new, if we keep track of the entry and exit directions.

Figure 3.6 gives a study of all possible patterns. We assume that the turtle has entered via the bottom edge, i.e. the previously visited neighbour is assumed to be below the current square. We then have the following variants:

- The turtle can enter at the left-hand or at the right-hand corner of the entry edge.
- After entry, the turtle can follow a clockwise or counter-clockwise Hilbert pattern.
- At the exit corner, the Hilbert curve can exit via two different edges (but not via the entry edge, as no cell is visited twice).

Not all combinations are possible, so we are left with six different patterns, which are illustrated in Fig. 3.6. For each scheme, the refinement scheme is illustrated. Note that the scheme is uniquely defined for all patterns. As the entry and exit conditions are explicitly considered, the refinement scheme particularly illustrates the contiguity of the Hilbert iteration: for the resulting grammars it would be straightforward to prove that they lead to contiguous iterations. In the proof of continuity of the Hilbert curve, this will proof the claim that two adjacent intervals are always mapped to adjacent subsquares.

With our experience from the previous sections, it is no longer difficult to derive a grammar from Fig. 3.6:

- We have six non-terminals: $\{H, L, B, E, R, T\}$, with start symbol H.
- We have three terminals: {↑, ↰, ↱}.

The traffic signs used as terminals illustrate the commands *forward*, *turn left*, and *turn right*. To determine the productions of the grammar, we have to define exactly where a pattern should start and end.

Grammar No. 1

For our first version of the grammar, we define that a patterns starts right after the turtle has entered a subsquare, and stops after it has entered the next subsquares – i.e. the transfer steps between the squares are included in the patterns. We thus obtain productions that do not contain any terminal characters:

$$\begin{array}{rclcrcl} H & \longleftarrow & B\,H\,T\,L & \qquad & B & \longleftarrow & H\,R\,L\,T \\ L & \longleftarrow & H\,R\,L\,E & & E & \longleftarrow & R\,H\,T\,L \\ T & \longleftarrow & R\,H\,T\,B & & R & \longleftarrow & E\,R\,L\,T \end{array}$$

Naturally, such a grammar does not lead to a traversal algorithm, because there are no movements involved. Hence, we need to add terminal productions, that draw the basic patterns:

$H \longleftarrow$ ↑ ↱ ↑ ↱ ↑ ↰ ↑ $\qquad B \longleftarrow$ ↱ ↑ ↰ ↑ ↰ ↑ ↱ ↑
$L \longleftarrow$ ↱ ↑ ↰ ↑ ↰ ↑ ↑ $\qquad E \longleftarrow$ ↰ ↑ ↱ ↑ ↱ ↑ ↰ ↑
$T \longleftarrow$ ↰ ↑ ↱ ↑ ↱ ↑ ↑ $\qquad R \longleftarrow$ ↑ ↰ ↑ ↰ ↑ ↱ ↑

Again, we have to modify the derivation rule: as usual, all non-terminals need to be replaced simultaneously; in addition, we either have to use the non-terminal productions for all non-terminals (recursion), or use the terminal productions for all non-terminals (leaf-case).

Grammar No. 2

In Grammar no. 1, we notice that the last step of each terminal production is always a *forward* command. This is, of course, the transfer step between two subsquares. If we remove this transfer step from the terminal productions, and insert it into the non-terminal productions, instead, we obtain the following productions:

$H \longleftarrow B$ ↑ H ↑ T ↑ L
$B \longleftarrow H$ ↑ R ↑ L ↑ T
$L \longleftarrow H$ ↑ R ↑ L ↑ E
$E \longleftarrow R$ ↑ H ↑ T ↑ L
$T \longleftarrow R$ ↑ H ↑ T ↑ B
$R \longleftarrow E$ ↑ R ↑ L ↑ T

$H \longleftarrow$ ↑ ↱ ↑ ↱ ↑ ↰
$B \longleftarrow$ ↱ ↑ ↰ ↑ ↰ ↑ ↱
$L \longleftarrow$ ↱ ↑ ↑ ↑ ↑ ↑
$E \longleftarrow$ ↑ ↑ ↱ ↑ ↱ ↑ ↑
$T \longleftarrow$ ↑ ↑ ↱ ↑ ↱ ↑
$R \longleftarrow$ ↑ ↑ ↑ ↑ ↑ ↱

(with the same additional derivation rules as for Grammar no. 1).

From these productions, we can see more clearly that our turtle will never do two rotations without an intermediate forward step. As two non-terminals are always separated by a forward move, turns that result from different non-terminals cannot occur with one immediately following after another. In the terminal production, we can also easily confirm that the turtle will never do more than one subsequent turn.

Grammar No. 3

If we examine the non-terminal and terminal productions in Grammar no. 2, we notice that for each non-terminal, the sequence of *forward* commands is identical for the non-terminal and the terminal production. Moreover, it turns out that the non-terminals L and R are always replaced by a ↰ in the terminal productions, whereas the non-terminals H and T are always replaced by a ↱. B and E are simply left away in the terminal productions. If we compare this with Fig. 3.6, we notice that the patterns L and R indeed lead to a left turn, while H and T lead to a right turn of the turtle. For B and E, the turtle basically runs straight across the subsquare.

We can therefore simplify our productions by using ϵ-productions for B and E:

Algorithm 3.5: 2D turtle-based Hilbert traversal

```
Procedure H(depth) begin
    if depth = 0 then
        execute(...);
        turnright();
    else
        B(depth-1); forward();
        H(depth-1); forward();
        T(depth-1); forward();
        L(depth-1);
    end
end
```

$$\begin{array}{lcl@{\qquad\qquad}lcl}
H & \longleftarrow & B \uparrow H \uparrow T \uparrow L & H & \longleftarrow & \Rsh \\
B & \longleftarrow & H \uparrow R \uparrow L \uparrow T & B & \longleftarrow & \epsilon \\
L & \longleftarrow & H \uparrow R \uparrow L \uparrow E & L & \longleftarrow & \Lsh \\
E & \longleftarrow & R \uparrow H \uparrow T \uparrow L & E & \longleftarrow & \epsilon \\
T & \longleftarrow & R \uparrow H \uparrow T \uparrow B & T & \longleftarrow & \Rsh \\
R & \longleftarrow & E \uparrow R \uparrow L \uparrow T & R & \longleftarrow & \Lsh
\end{array}$$

Note that the terminal productions of Grammar no. 2 can be reconstructed from this grammar by applying the non-terminal productions followed by the terminal productions.

Turtle-Based Traversals

Again, we can turn our grammars into traversal algorithms for 2D data structures – see Algorithm 3.5 for an implementation based on Grammar no. 3. The new traversal algorithms correspond to turtle graphics, as it was used by early graphics programming languages: the turtle is allowed to move forward, or do specific left or right turns. We will thus call this traversal a turtle-based traversal, and refer to the earlier traversal algorithms as plotter-based traversals.

There are a couple of interesting differences between the turtle-based and plotter-based traversals:

1. During the turtle-based traversals, we can easily determine the previously visited element, and also the following element. This is not easy to achieve in the plotter-based traversal.
2. In contrast, the turtle-based traversal can no longer directly determine the left or right neighbour, as we do not have an absolute orientation. However, this is quite easily cured by storing the current direction.

3. For the traversal of a 2D array, the implementation of the basic moves of the plotter-based traversal corresponds to simple increment and decrement operations on the array indices. In addition, only one index has to be incremented or decremented in each step. This is a bit simpler than for the turtle-based traversal, where the increments have to be stored somehow.

Moreover, the relative orientation used in the turtle-based grammars will be helpful to describe the $\beta\Omega$-curve introduced in Sect. 7.4. We will also use it for the traversal of quadtree and octree structures in Chap. 14.

References and Further Readings

While the productions used to describe the iterations of space-filling curves are all compatible with context-free grammars of the Chomsky hierarchy, the additional rule to expand all non-terminals simultaneously in a given string turns our grammars into so-called *L-systems*. L-systems where introduced by Lindenmayer [162] to model the growth of organisms, where multiple, but similar processes happen at the same time at different locations of the organism. The first description of space-filling curves via L-systems (with chain code interpretation) was given by Siromoney and Subramanian [246]. Prusinkiewicz [221, 223] suggested the interpretation via turtle graphics, and presented L-systems for 2D Hilbert, Peano, and Sierpinski curves (see also Exercise 3.5), as well as for certain fractal curves including the Gosper curve (see Sect. 7.5). An instructive introduction to this modelling approach is given in [222].

Gips [100] used the Koch snowflake (see Sect. 5.4), as well as the 2D Hilbert and Peano curve as examples for the use of *shape grammars*. His grammar for the Hilbert curve is very similar to Fig. 3.6. He introduced *serial* and *parallel* grammars, which only differ in the application of the production rules – parallel application of the rules being identical to our requirement of replacing all non-terminal symbols at once, and thus leading to L-systems. Serial shape grammars may lead to adaptive curves, as we will discuss in Chap. 9.

A characterisation of the curve patterns via symbols, as in Fig. 3.1, was already used by Borel [45], in 1949. He set up tables to determine the patterns of the Hilbert curve in successive subdomains, and described an algorithm to compute the image points of parameters, based on their quaternary representation. Our Algorithm 3.1 is practically identical to the one given by Wirth [269, 270], who also provided the respective algorithm for the Sierpinski curve (see Chap. 6). He also adopted the corresponding production rules to describe the basic patterns and construction of the respective iterations. Goldschlager [101] modified Wirth's algorithm into a compact recursive algorithms by coding the patterns into a set of four parameters that reflect the orientation of the curve in the current subsquare. Witten and Wyhill [274] noted the similarity of the resulting algorithm to respective recursive algorithms for "experimental graphics languages", which where also used to generate so-called *dragon curves* (see Sect. 5.4).

Griffiths [110] introduced table-based implementations of the grammar-based algorithm. His algorithms generate the plotter steps to draw the respective curves. In [111], he extended his study by an overview on curves that are based on subdividing squares into 4 × 4 or 5 × 5 subsquares. A table-based index computation for the Hilbert curve was also given by Bandou and Kamata [26]. In 2005, Jin and Mellor-Crummey [141] described a table-based framework that uses tables to efficiently implement grammar-based traversal algorithms. By recording relative positions in the tables, they also describe algorithms to compute the indexing for different space-filling curves. They demonstrated that their table-based implementation leads to faster computation of space-filling mapping and traversals than algorithms based on an arithmetisation of space-filling curves (see Chap. 4 and the references therein) – which also demonstrates that the runtime of such algorithms depends to a large degree on the available hardware.

What's next?

- While grammars are helpful to design space-filling-curve traversals, it is difficult to use them for retrieving individual elements. For that problem, we require the space-filling curve as a parameter-to-image-point mapping – which will be the topic of the next chapter.
- If you are wondering, how grammars and resulting traversals can be used for adaptive quadtree or octree grids, as indicated in Chap. 1, then take a detour to Chap. 9 (in particular, Sect. 9.2.2).

Exercises

3.1. In analogy to the grammars for the Hilbert and Peano curve, derive a grammar that describes the iterations of the Peano-Meander curve, as given in Fig. 2.8b on page 27.

3.2. In the same way, derive a grammar that describes the iterations of Moore's version of the Hilbert curve (see Sect. 2.3.6).

3.3. Extend Algorithm 3.2 such that it performs a given operation on a matrix A stored in Hilbert order – such as multiplying all elements of A with a scalar, or computing the matrix-vector product Ax with a specified vector x.

3.4. In Sect. 3.4, we introduced turtle-based traversals for the Hilbert curve. Derive a corresponding turtle-based traversal (and respective grammar) for the Peano curve.

3.5. The number of non-terminals used for the turtle grammars, as in Sect. 3.4, can be reduced, if we allow the turtle to perform multiple rotation operations between its steps. Show that only two non-terminals are then sufficient. (The L systems presented by Prusinkiewicz [221] followed such an approach.)

Chapter 4
Arithmetic Representation of Space-Filling Curves

Up to now, we have dealt with the finite iterations of the Hilbert and Peano curve, only. However, these can only offer an approximate impression of the "infinite" curves. Moreover, we are not yet able to compute the corresponding Hilbert or Peano mapping, i.e. to compute the image point for a given parameter. The grammar representations of space-filling curves are not directly suitable for this purpose, as they always generate the iterations as a whole.

Hence, in this chapter we will describe a system to describe Hilbert curves, as well as Peano curves and many similar curves, in an arithmetic way that allows the efficient computation of the respective mappings. While the grammar representation lead to algorithms to traverse data structures in space-filling-curve order, arithmetic mappings (and their inverses) will allow us to reference and dereference individual elements of these data structures.

4.1 Arithmetic Representation of the Hilbert Mapping

Let us first remember the following basic construction principles of the Hilbert curve:

1. A given parameter is approximated by nested intervals: each interval is one of the four quarters of its predecessor, starting with the parameter interval $\mathcal{I}$.
2. The target point is approximated by nested subsquares: each of the subsquares is again filled by a scaled-down, transposed, and rotated or reflected Hilbert curve section.

The following *arithmetisation* of the Hilbert curve turns these two ideas into a formal, mathematical description of the Hilbert mapping.

M. Bader, *Space-Filling Curves*, Texts in Computational Science and Engineering 9, DOI 10.1007/978-3-642-31046-1_4, © Springer-Verlag Berlin Heidelberg 2013

Quaternary Representation of Parameters

For the required nested intervals, the representation of the parameters as quaternaries proves to be valuable (see Exercise 4.1 for a short discussion of quaternary representations). For that purpose, we consider the typical boundaries of the nested intervals in quaternary representation:

$$\left[0, \frac{1}{4}\right] = [0_4.0, 0_4.1], \qquad \left[\frac{1}{4}, \frac{2}{4}\right] = [0_4.1, 0_4.2],$$
$$\left[\frac{2}{4}, \frac{3}{4}\right] = [0_4.2, 0_4.3], \qquad \left[\frac{3}{4}, 1\right] = [0_4.3, 1_4.0].$$

The quaternary digits directly reflect the numbering of the subsquares in each iteration. For example, for the parameter $t = \frac{2}{5}$ we obtain the nested intervals

$$[0, 1], [0_4.1, 0_4.2], [0_4.12, 0_4.13], [0_4.121, 0_4.122], \ldots,$$

which correspond to the quaternary $\frac{2}{5} = 0_4.121212\ldots$

Recursively Mapping Subsquares to the Unit Square

The second main idea to construct the Hilbert curve is its self-similarity: the Hilbert curve consists of four identical sub-curves, each of which fills one of the four subsquares. In addition, each of these sub-curves is a scaled-down, rotated, and translated copy of the original Hilbert curve.

Following a classical recursive approach, we assume that we are able to compute the curve, i.e. the image point of a given parameter, for the scaled-down Hilbert curves in the subsquares. Such an image point will be computed relative to the unit square $[0, 1]^2$. Hence, we need to compute its relative position in the respective subsquare, subject to the scaling, rotation, and translation of the Hilbert curve in this section.

For each subsquare, we therefore require a transformation operator that maps the unit square into the correct subsquare and performs the necessary transformations. Following Fig. 4.1, we will denote these operators as H_0, H_1, H_2, and H_3. The quaternary digits will tell us the intervals that contain the parameters and therefore determine the subsquares of the images and, as a consequence, which operator has to be applied.

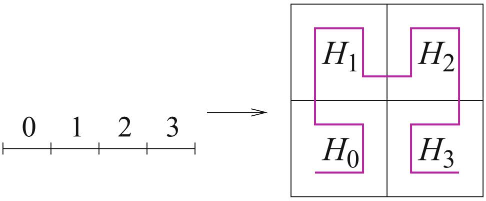


Fig. 4.1 Intervals and corresponding transformation operators $H_0, \dots, H_3$ (compare Fig. 2.4 on page 21)

In matrix-vector notation, the four operators are:

$$H_0 := \begin{pmatrix} 0 & \frac{1}{2} \\ \frac{1}{2} & 0 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \qquad H_1 := \begin{pmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 0 \\ \frac{1}{2} \end{pmatrix}$$

$$H_2 := \begin{pmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} \frac{1}{2} \\ \frac{1}{2} \end{pmatrix} \qquad H_3 := \begin{pmatrix} 0 & -\frac{1}{2} \\ -\frac{1}{2} & 0 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 1 \\ \frac{1}{2} \end{pmatrix}$$

All operators H_i first scale the argument vector (i.e. the image point) by a factor of $\frac{1}{2}$, as all subsquares have half the side length of the original square. In addition, each operator performs a combination of rotations, reflections, and translations:

- Operator H_0 performs a reflection at the main diagonal (swapping x- and y-coordinates), which corresponds to the required clockwise 90° turn plus change of orientation.
- The operators H_1 and H_2 retain the orientation of the curve. Thus, no rotation or reflections are necessary, and the operation matrix only performs a scaling. In addition, the scaled-down curve is translated into the two respective subsquares. H_1 translates the curve by $\frac{1}{2}$ in y-direction, i.e. into the top-left subsquare. Likewise, H_2 diagonally translates the curve into the top-right subsquare.
- Operator H_3 requires the exchange of the x- and y-coordinates (similar to operator H_0), but in addition a reflection in both x- and y-direction – we therefore scale with a negative factor, $-\frac{1}{2}$). Note that the reflection along the vertical axes is required to obtain the correct orientation of the curve: the curve enters the subsquare (i.e. starts) in the top-right corner and ends in the lower-right corner. The position of the origin of the sub-curve in the top-right corner also determines the translation vector of the curve, $\begin{pmatrix} 1 \\ \frac{1}{2} \end{pmatrix}$.

4.2 Calculating the Values of *h*

With the given operators $H_0, \dots, H_3$, the function values $h(t)$ are calculated via the following three steps:

1. Compute the quaternary representation of the parameter: $t = 0_4.q_1q_2q_3q_4\dots$ Then, parameter t lies in the q_1-th interval and $h(t)$ in the q_1-th subsquare.

2. In the q_1-th subsquare, parameter t corresponds to the local parameter $\tilde{t} = 0_4.q_2q_3q_4\ldots$ – with deleted first digit in the quaternary representation. We therefore compute $h(\tilde{t})$, which is the image of $\tilde{t}$ in the q_1-th subsquare relative to the scaled-down and rotated Hilbert curve in that subsquare.
3. Transform the local coordinates of $h(\tilde{t})$ (relative to the subsquare) into the coordinates in the original square. This transformation is performed by applying the operator H_{q_1} to $h(\tilde{t})$.

Hence, the main steps for arithmetisation of the Hilbert curve may be combined into the following recursive equation:

$$h(0_4.q_1q_2q_3q_4\ldots) = H_{q_1} \circ h(0_4.q_2q_3q_4\ldots). \tag{4.1}$$

By successively applying the recursion on h, we obtain

$$\begin{aligned} h(0_4.q_1q_2q_3q_4\ldots) &= H_{q_1} \circ h(0_4.q_2q_3q_4\ldots) \\ &= H_{q_1} \circ H_{q_2} \circ h(0_4.q_3q_4\ldots) \\ &= \ldots \end{aligned}$$

How is this recursion terminated? To answer this question, we first examine the case of finite quaternaries, where all quaternary digits after a certain q_n are 0. We obtain

$$\begin{aligned} h(0_4.q_1q_2\ldots q_n) &= h(0_4.q_1q_2\ldots q_n000\ldots) \\ &= H_{q_1} \circ H_{q_2} \circ \ldots \circ H_{q_n} \circ \underbrace{h(0_4.000\ldots)}_{=h(0)}. \end{aligned}$$

From the construction of the Hilbert curve, we seem to know that $h(0) = (0, 0)$. But is this consistent with our outlined arithmetisation? For $t = 0 = 0_4.000\ldots$, the recursion Eq. (4.1) leads to

$$h(0_4.0000\ldots) = H_0 \circ h(0_4.000\ldots) \qquad \Leftrightarrow \qquad h(0) = H_0 \circ h(0).$$

$h(0)$ therefore needs to be a fixpoint of H_0. Indeed, $h(0) = (0, 0)$ is the only fixpoint of H_0. For finite quaternaries $t = 0_4.q_1q_2q_3\ldots q_n$, we can thus compute $h(t)$ by

$$h(0_4.q_1q_2q_3\ldots q_n) = H_{q_1} \circ H_{q_2} \circ H_{q_3} \circ \cdots \circ H_{q_n} \begin{pmatrix} 0 \\ 0 \end{pmatrix}. \tag{4.2}$$

For infinite quaternaries t, this is generalised to

$$h(0_4.q_1q_2q_3\ldots) = \lim_{n\to\infty} H_{q_1} \circ H_{q_2} \circ H_{q_3} \circ \cdots \circ H_{q_n} \begin{pmatrix} 0 \\ 0 \end{pmatrix}. \tag{4.3}$$

Example: Computation of $h\left(\frac{1}{4}\right)$ and $h\left(\frac{1}{8}\right)$

To compute the Hilbert image point of the parameters $t = \frac{1}{4}$ and $t = \frac{1}{8}$, we first need to determine their quaternary representation. We obtain:

$$\frac{1}{4} = 0_4.1 \qquad \text{and} \qquad \frac{1}{8} = 0_4.02$$

According to Eq. (4.2), we get:

$$h\left(\frac{1}{4}\right) = H_1 \begin{pmatrix} 0 \\ 0 \end{pmatrix} = \begin{pmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{pmatrix} \begin{pmatrix} 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ \frac{1}{2} \end{pmatrix} = \begin{pmatrix} 0 \\ \frac{1}{2} \end{pmatrix}.$$

This is consistent with intuition: after having covered one quarter of the parameter interval, the Hilbert curve should have visited the entire lower-left quarter of the unit square. Hence, $h\left(\frac{1}{4}\right)$ should be the connection point between first and second subsquare. From the approximating polygon, we know that this should be the point $\left(0, \frac{1}{2}\right)$.

In the same way, we can compute the value of $h\left(\frac{1}{8}\right)$:

$$\begin{aligned} h\left(\frac{1}{8}\right) &= H_0 \circ H_2 \begin{pmatrix} 0 \\ 0 \end{pmatrix} = H_0 \left(\begin{pmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{pmatrix} \begin{pmatrix} 0 \\ 0 \end{pmatrix} + \begin{pmatrix} \frac{1}{2} \\ \frac{1}{2} \end{pmatrix} \right) \\ &= H_0 \begin{pmatrix} \frac{1}{2} \\ \frac{1}{2} \end{pmatrix} = \begin{pmatrix} 0 & \frac{1}{2} \\ \frac{1}{2} & 0 \end{pmatrix} \begin{pmatrix} \frac{1}{2} \\ \frac{1}{2} \end{pmatrix} = \begin{pmatrix} \frac{1}{4} \\ \frac{1}{4} \end{pmatrix}. \end{aligned}$$

Compare this with the second approximating polygon of the Hilbert curve (see Fig. 2.1 on page 19). The point $h\left(\frac{1}{8}\right) = h\left(\frac{2}{16}\right)$ is where the Hilbert curve transfers from the second to the third subsquare (out of 16) of the respective level – which is the point $\left(\frac{1}{4}, \frac{1}{4}\right)$.

An Algorithm to Compute the Hilbert Mapping

An algorithm to compute the values of the mapping h has to perform two separate tasks:

1. Computation of the quaternary representation: following the equation

$$4 \cdot 0_4.q_1q_2q_3q_4 \ldots = (q_1.q_2q_3q_4 \ldots)_4,$$

   we obtain the quaternary digits by repeatedly multiplying the parameters by 4, and cutting off the integer part.

Function hilbert – compute the Hilbert mapping (fixed number of digits)

```
Function hilbert (t, depth)
    Parameter: t: index parameter, $t \in [0, 1]$
               depth: depth of recursion (equiv. to number of quaternary digits)
begin
    if depth = 0 then
        return (0,0)
    else
        // compute next quaternary digit in q
        q := floor (4*t);
        r := 4*t - q;
        // recursive call to h()
        (x,y) := hilbert (r,depth-1);
        // apply operator $H_q$
        switch q do
            case 0: return ( y/2, x/2);
            case 1: return ( x/2, y/2 + 0.5);
            case 2: return ( x/2+0.5, y/2+0.5);
            case 3: return ( 1.0-y/2, 0.5-x/2);
        endsw
    end
end
```

2. Application of the respective operators H_q in the correct order – in a recursive algorithm, we can simply use Eq. (4.1) for that purpose.

A simple termination criterion for the algorithm is given by the number of quaternary digits considered in the parameter. In a recursive algorithm, this corresponds to the number of recursive applications of an operator H_q. Function `hilbert` is a recursive implementation of the final algorithm.

As an alternative, we can terminate the recursion once a given accuracy ϵ is obtained. If we interprete ϵ to be the side length of the current subsquare with the nested 2D-intervals (which contains the desired image point), then $\epsilon = 2^{-depth}$, and we can transfer Function `hilbert` into Function `hilbertEps`.

4.3 Uniqueness of the Hilbert Mapping

In this section, we will come back to a question raised during the definition and construction of the Hilbert curve:

> Are the function values of h independent of the choice of quaternary representation in the sense that
>
> $$h(0_4.q_1 \dots q_n) = h(0_4.q_1 \dots q_{n-1}(q_n - 1)333\dots), \qquad q_n \neq 0\,?$$

Function hilbertEps – compute the Hilbert mapping (prescribed accuracy)

```
Function hilbertEps (t, eps)
    Parameter: t: index parameter, $t \in [0, 1]$
               eps: required precision (of point coordinates)
begin
    if eps > 1 then
        return (0,0)
    else
        // compute next quaternary digit in q
        q := floor (4*t);
        r := 4*t - q;
        // recursive call to h()
        (x,y) := hilbertEps (r,2*eps);
        // apply operator $H_q$
        switch q do
            case 0: return ( y/2, x/2);
            case 1: return ( x/2, y/2 + 0.5);
            case 2: return ( x/2+0.5, y/2+0.5);
            case 3: return ( 1.0-y/2, 0.5-x/2);
        endsw
    end
end
```

Note that we may disregard the case $q_n = 0$, as then $0_4.q_1 \dots q_n = 0_4.q_1 \dots q_{n-1}$ and our problem occurs at the previous digit.

Note that the two different quaternary representations correspond to two different sequences of nested intervals – as we discussed in Sect. 2.3.4. The arithmetisation, however, provides us with a mathematical tool to prove uniqueness independent of the continuity argument. Our proof works in two steps:

1. Compute the limit $\lim_{n\to\infty} H_3^n$, in particular $\lim_{n\to\infty} H_3^n \begin{pmatrix}0\\0\end{pmatrix}$.
2. For all $q_n = 1, 2, 3$, show that $H_{q_n} \circ \begin{pmatrix}0\\0\end{pmatrix} = H_{q_n-1} \circ \lim_{n\to\infty} H_3^n \begin{pmatrix}0\\0\end{pmatrix}$.

Compute h for Infinite Quaternaries

The computation of $\lim_{n\to\infty} H_3^n$ works as an example for how to compute h for infinite quaternaries, in general. Though, for our proof, we only need the value

$$h(0_4.3333\dots) = H_3 \circ H_3 \circ \dots \begin{pmatrix}0\\0\end{pmatrix} = \lim_{n\to\infty} H_3^n \begin{pmatrix}0\\0\end{pmatrix}.$$

We first write the operator H_3 in the form

$$H_3\colon \begin{pmatrix} x \\ y \end{pmatrix} \to \underbrace{\begin{pmatrix} 0 & -\frac{1}{2} \\ -\frac{1}{2} & 0 \end{pmatrix}}_{=:A_3} \begin{pmatrix} x \\ y \end{pmatrix} + \underbrace{\begin{pmatrix} 1 \\ \frac{1}{2} \end{pmatrix}}_{:=b_3} \qquad \text{i.e.} \qquad H_3\colon v \to A_3 v + b_3.$$

Based on this representation, we can compute the effect of H_3^n:

$$\begin{aligned} H_3^2 v &= A_3(A_3 v + b_3) + b_3 = A_3^2 v + A_3 b_3 + b_3 \\ H_3^3 v &= A_3(A_3^2 v + A_3 b_3 + b_3) + b_3 = A_3^3 v + A_3^2 b_3 + A_3 b_3 + b_3 \\ &\;\;\vdots \\ H_3^n v &= A_3^n v + A_3^{n-1} b_3 + \ldots + A_3 b_3 + b_3 \end{aligned}$$

The term $A_3^{n-1} b_3 + \ldots + A_3 b_3 + b_3$ is more or less a geometric series, so we can use a well-known trick:

$$\begin{aligned} (I - A_3)\left(A_3^{n-1} b_3 + \ldots + A_3 b_3 + b_3\right) &= A_3^{n-1} b_3 + \ldots + A_3 b_3 + b_3 \\ &\quad -A_3^n b_3 - A_3^{n-1} b_3 - \ldots - A_3 b_3 \\ &= b_3 - A_3^n b_3 = \left(I - A_3^n\right) b_3 \end{aligned}$$

Therefore, we obtain

$$A_3^{n-1} b_3 + \ldots + A_3 b_3 + b_3 = (I - A_3)^{-1}\left(I - A_3^n\right) b_3,$$

and, as $\lim\limits_{n\to\infty} A_3^n = 0$, we get:

$$\begin{aligned} \lim_{n\to\infty} H_3^n \begin{pmatrix} 0 \\ 0 \end{pmatrix} &= \lim_{n\to\infty} \left(A_3^n \begin{pmatrix} 0 \\ 0 \end{pmatrix} + A_3^{n-1} b_3 + \ldots + A_3 b_3 + b_3 \right) \\ &= \lim_{n\to\infty} \left((I - A_3)^{-1} (I - \underbrace{A_3^n}_{\to 0}) b_3 \right) = (I - A_3)^{-1} b_3. \end{aligned}$$

Hence, we can determine the function value $h(1) = h\left(0_4.3333\ldots\right) =: \binom{x}{y}$ by solving the system of equations

$$(I - A_3)\begin{pmatrix} x \\ y \end{pmatrix} = b_3 \qquad \Leftrightarrow \qquad \begin{pmatrix} 1 & \frac{1}{2} \\ \frac{1}{2} & 1 \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 1 \\ \frac{1}{2} \end{pmatrix}.$$

As the right hand side is identical to the first column of the system matrix, we can directly read the solution as $x = 1$ and $y = 0$. Therefore,

$$h\left(0_4.3333\ldots\right) = h(1) = \begin{pmatrix} 1 \\ 0 \end{pmatrix},$$

which is what we had to expect – the Hilbert curve ends in the lower-right corner of the unit square.

Uniqueness of the Hilbert Mapping

To prove uniqueness, we still have to show that

$$H_{q_n}\begin{pmatrix} 0 \\ 0 \end{pmatrix} = H_{q_n - 1}\begin{pmatrix} 1 \\ 0 \end{pmatrix} \qquad \text{for all } n = 1, 2, 3.$$

We will demonstrate this proof only for $n = 1$:

$$\begin{aligned} H_1\begin{pmatrix} 0 \\ 0 \end{pmatrix} &= \begin{pmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{pmatrix}\begin{pmatrix} 0 \\ 0 \end{pmatrix} + \begin{pmatrix} 0 \\ \frac{1}{2} \end{pmatrix} = \begin{pmatrix} 0 \\ \frac{1}{2} \end{pmatrix} \qquad \text{and} \\ H_0\begin{pmatrix} 1 \\ 0 \end{pmatrix} &= \begin{pmatrix} 0 & \frac{1}{2} \\ \frac{1}{2} & 0 \end{pmatrix}\begin{pmatrix} 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 0 \\ \frac{1}{2} \end{pmatrix} \quad \text{q.e.d.} \end{aligned}$$

Hence, the computation of the Hilbert function is independent of the choice of quaternary representation, and thus also of the choice of nested intervals.

4.4 Computation of the Inverse: Hilbert Indices

Up to now, we computed the image points $h(t)$ for a given parameter t. However, to compute the inverse problem is at least of equal relevance in practice:

For a given point $(x, y) \in \mathcal{Q}$, find a parameter t, such that $h(t) = (x, y)$.

In an application context, this task is equivalent to finding the memory location (i.e., the index) of a given tuple (x, y).

Uniqueness of the Inverse Mapping

First, we remember a previous result: the Hilbert curve (as well as the Peano curve) is surjective, but not bijective! Hence, there are points (x, y) that are images of multiple parameters t. Therefore:

- An inverse mapping h^{-1} does not exist in the strict sense.
- We can only define and compute a mapping $\bar{h}^{-1}$ that is technically forced to be unique.
- We will call the parameter $\bar{h}^{-1}(x, y)$ the *Hilbert index* of the point (x, y).

The computation of the Hilbert index follows the same recursive principle as the computation of the Hilbert mapping, but works in opposite direction:

1. We first determine the subsquare that contains (x, y).
2. Using the inverses of the operators $H_0, \dots, H_3$, we can map (x, y) into the unit square. We obtain the point $(\tilde{x}, \tilde{y})$, which corresponds to the relative position of (x, y) in its subsquare.
3. Via a recursive call, we compute a parameter $\tilde{t}$ that is mapped to $(\tilde{x}, \tilde{y})$.
4. Depending on the subsquare, we compute the final Hilbert index t from the relative index $\tilde{t}$.

Determining the Inverse Operators of $H_0, \dots, H_3$

We obtain the inverse operators of $H_0, \dots, H_3$ by solving the respective transforms for the source variables. For example, for H_0 we obtain:

$$\begin{pmatrix} x \\ y \end{pmatrix} = H_0 \begin{pmatrix} \tilde{x} \\ \tilde{y} \end{pmatrix} = \begin{pmatrix} \frac{1}{2}\tilde{y} \\ \frac{1}{2}\tilde{x} \end{pmatrix} \quad \Rightarrow \quad \begin{pmatrix} \tilde{x} \\ \tilde{y} \end{pmatrix} = \begin{pmatrix} 2y \\ 2x \end{pmatrix}$$

Analogous computation for H_1, H_2, and H_3 leads to the following inverse operators:

$$H_0^{-1} := \begin{pmatrix} x \\ y \end{pmatrix} \to \begin{pmatrix} 2y \\ 2x \end{pmatrix} \qquad H_1^{-1} := \begin{pmatrix} x \\ y \end{pmatrix} \to \begin{pmatrix} 2x \\ 2y - 1 \end{pmatrix}$$

$$H_2^{-1} := \begin{pmatrix} x \\ y \end{pmatrix} \to \begin{pmatrix} 2x - 1 \\ 2y - 1 \end{pmatrix} \qquad H_3^{-1} := \begin{pmatrix} x \\ y \end{pmatrix} \to \begin{pmatrix} -2y + 1 \\ -2x + 2 \end{pmatrix}$$

An Algorithm to Compute the Hilbert Index

The (technically unique) inverse mapping $\bar{h}^{-1}$ is then obtained by inverting the computation of the Hilbert mapping step by step:

1. For a given point (x, y), determine the subsquare depending on whether $x <> \frac{1}{2}$ and $y <> \frac{1}{2}$, and read the number $q \in \{0, \ldots, 3\}$ of this subsquare according to the scheme

1	2
0	3

   The ambiguous cases $x = \frac{1}{2}$ and/or $y = \frac{1}{2}$ have to be uniquely classified as either $<$ or $>$ in order to achieve *uniqueness* of the inverse mapping.
2. With the respective inverse operator H_q^{-1}, we then determine the relative position of (x, y) in the subsquare q, and obtain $(\tilde{x}, \tilde{y}) := H_q^{-1}(x, y)$
3. We (recursively!) compute the Hilbert index $\tilde{t}$ of the transformed point $(\tilde{x}, \tilde{y})$ in subsquare q: $\tilde{t} := \bar{h}^{-1}(\tilde{x}, \tilde{y})$.
4. The Hilbert index of (x, y) is then obtained as $t := \frac{1}{4}\left(q + \tilde{t}\right)$.

We still need to add a termination criterion to this recursive scheme, which can either be to consider a fixed number of recursions (which corresponds to a fixed number of digits), or to demand a certain accuracy ϵ for the computed Hilbert index. With every recursive call, the accuracy ϵ may be multiplied by 4, as the transformation from subsquare to original square scales the respective parameter interval by a factor of 4. If, after several steps of recursion, $\epsilon > 1$, we may deliver any value in $[0, 1]$ as result. We thus obtain Function `hilbIndex` to compute the Hilbert index up to a given accuracy `eps`.

4.5 Arithmetisation of the Peano Curve

The arithmetisation of the Peano curve works in exactly the same way as for the Hilbert curve. Instead of partitioning subsquares and intervals into four parts in each step, the Peano curve now requires partitioning into nine parts. As a consequence, we base our computations on representing the parameter t in the "nonal system", i.e. $t = 0_9.n_1n_2n_3n_4\ldots$. Likewise, we determine operators $P_0, \ldots, P_8$, such that

$$p(0_9.n_1n_2n_3n_4\ldots) = P_{n_1} \circ P_{n_2} \circ P_{n_3} \circ P_{n_4} \circ \cdots \begin{pmatrix} 0 \\ 0 \end{pmatrix}.$$

The following scheme lists the operators P_n such that their geometrical position reflects the corresponding subsquares in the recursive construction of the Peano curve:

Function hilbIndex – compute the Hilbert index (prescribed accuracy)

```
Function hilbIndex(x,y,eps)
    Parameter: x,y: coordinates of image point, $x, y \in [0,1]^2$
               eps: required precision (of index parameter)
begin
    if eps > 1 then return 0;
    if x < 0.5 then
        if y < 0.5 then
            return ( 0 + hilbIndex(2*y, 2*x, 4*eps) )/4;
        else
            return ( 1 + hilbIndex(2*x, 2*y - 1, 4*eps) )/4;
        end
    else
        if y ≥ 0.5 then
            return ( 2 + hilbIndex(2*x-1, 2*y - 1, 4*eps) )/4;
        else
            return ( 3 + hilbIndex(1-2*y, 2-2*x, 4*eps) )/4;
        end
    end
end
```

$$P_2\begin{pmatrix}x\\y\end{pmatrix} = \begin{pmatrix}\frac{1}{3}x+0\\ \frac{1}{3}y+\frac{2}{3}\end{pmatrix} \quad P_3\begin{pmatrix}x\\y\end{pmatrix} = \begin{pmatrix}\frac{1}{3}x+\frac{1}{3}\\ -\frac{1}{3}y+1\end{pmatrix} \quad P_8\begin{pmatrix}x\\y\end{pmatrix} = \begin{pmatrix}\frac{1}{3}x+\frac{2}{3}\\ \frac{1}{3}y+\frac{2}{3}\end{pmatrix}$$

$$P_1\begin{pmatrix}x\\y\end{pmatrix} = \begin{pmatrix}-\frac{1}{3}x+\frac{1}{3}\\ \frac{1}{3}y+\frac{1}{3}\end{pmatrix} \quad P_4\begin{pmatrix}x\\y\end{pmatrix} = \begin{pmatrix}-\frac{1}{3}x+\frac{2}{3}\\ -\frac{1}{3}y+\frac{2}{3}\end{pmatrix} \quad P_7\begin{pmatrix}x\\y\end{pmatrix} = \begin{pmatrix}-\frac{1}{3}x+1\\ \frac{1}{3}y+\frac{1}{3}\end{pmatrix}$$

$$P_0\begin{pmatrix}x\\y\end{pmatrix} = \begin{pmatrix}\frac{1}{3}x+0\\ \frac{1}{3}y+0\end{pmatrix} \quad P_5\begin{pmatrix}x\\y\end{pmatrix} = \begin{pmatrix}\frac{1}{3}x+\frac{1}{3}\\ -\frac{1}{3}y+\frac{1}{3}\end{pmatrix} \quad P_6\begin{pmatrix}x\\y\end{pmatrix} = \begin{pmatrix}\frac{1}{3}x+\frac{2}{3}\\ \frac{1}{3}y\end{pmatrix}$$

The operators need to scale the given point vectors by a factor of $\frac{1}{3}$, as in the Peano curve construction the subsquares are obtained by partitioning into 3×3 subsquares in each steps. In general, the operators are a little simpler than for the Hilbert curve:

- For the Peano curve, no operator requires a rotation or exchange of x- und y-coordinates. Reflections in the horizontal and/or the vertical direction are sufficient. As a result, the "switch-backs" of the curve are always in vertical direction.
- The operators P_0, P_2, P_6, and P_8 retain the orientation of the curve – here, we only require a scaling and translation to the new starting point.
- The operators P_1 and P_7 require reflection at the horizontal axis. Hence, the x-coordinate changes its sign.

Function peano – compute the Peano mapping (fixed number of digits)

```
Function peano (t, depth)
    Parameter: t: index parameter, $t \in [0, 1]$
               depth: depth of recursion (equiv. to number of digits)
begin
    if depth = 0 then
        return (0,0)
    else
        // compute the next digit
        q := floor (9*t);
        r := 9*t - q;
        (x,y) := peano (r,depth-1);
        switch q do
            case 0: return ( x/3, y/3 );
            case 1: return ( (-x+1)/3, (y+1)/3 );
            case 2: return ( x/3, (y+2)/3 );
            case 3: return ( (x+1)/3, (-y+3)/3 );
            case 4: return ( (-x+2)/3, (-y+2)/3 );
            case 5: return ( (x+1)/3, (-y+1)/3 );
            case 6: return ( (x+2)/3, y/3 );
            case 7: return ( (-x+3)/3, (y+1)/3 );
            case 8: return ( (x+2)/3, (y+2)/3 );
        endsw
    end
end
```

- In contrast, the operators P_3 and P_5 require reflection at the horizontal axis – and, hence, a change of sign of the y-coordinate.
- Finally, operator P_4 requires reflection both at the x- and y-axis (note the change of orientation).

Analogous to the Hilbert mapping, the arithmetisation of the Peano curve can be turned into a recursive function to compute the Peano mapping. See the algorithm given in Function `peano`.

4.6 Efficient Computation of Space-Filling Mappings

For all of the space-filling curves discussed in this book, we will be able to derive an arithmetisation as given for the Hilbert curve in Eq. (4.2):

$$h(0_4.q_1q_2q_3\dots q_n) = H_{q_1} \circ H_{q_2} \circ H_{q_3} \circ \cdots \circ H_{q_n} \circ h(0).$$

Straightforward recursive implementations, as used in the functions `hilbert`, `hilbertEps` or `peano`, however, are not necessarily efficient – the frequent recursive calls might sum up to a considerable computational overhead.

Function hilbertUnroll – efficient computation of the Hilbert mapping (with unrolling)

```
Function hilbertUnroll(t, depth)
    Parameter: t: index parameter, $t \in [0, 1]$
               depth: depth of recursion (equiv. to number of hex digits)
begin
    if depth = 0 then
        return (0,0)
    else
        // compute next hex digit in q
        q := floor(16*t);
        r := 16*t - q;
        // recursive call to h()
        (x,y) := hilbertUnroll(r,depth-1);
        // read operator H[q] from lookup table
        return H[q]( x, y );
    end
end
```

Such overheads can be reduced in a standard way by using recursion unrolling techniques, i.e. by combining two or more recursive calls into one.

4.6.1 Computing Hilbert Mappings via Recursion Unrolling

In our operator notation, we can thus write the Hilbert mapping in the form

$$h(0_4.q_1q_2q_3\ldots q_n) = (H_{q_1} \circ H_{q_2}) \circ (H_{q_3} \circ H_{q_4}) \circ \cdots \circ (H_{q_{n-1}} \circ H_{q_n}) \circ h(0).$$

Combining two quaternary digits into a hex digit, and setting $q_{12} := q_1q_2$, we obtain

$$h(0_4.q_{12}q_{34}\ldots q_{n-1,n}) = (H_{q_1q_2}) \circ (H_{q_3q_4}) \circ \cdots \circ (H_{q_{n-1},q_n}) \circ h(0), \qquad (4.4)$$

where the operators $H_{q_iq_j}$ are defined as $H_{q_iq_j} := H_{q_i} \circ H_{q_j}$. A recursive implementation of Eq. (4.4) requires only half the number of recursive calls – even more important, such an implementation also requires half the number of operator evaluations – check Exercise 4.6 or Sect. 4.6.2 to see that the operators stay linear and do not become more complicated.

As the hex digits q_{ij} now require 16 different operators, we pay for the reduction of operator executions by an increased number of different operators. In practice, we can store these operators in a respective table, which leads to an implementation as given in Function `hilbertUnroll`.

For a high-performance implementation, a couple of further optimisations to Function `hilbertUnroll` are possible:

- For number schemes that have a power of two as basis, the digits can be directly read from the binary representation of the input parameter, which saves the effort to compute the digits arithmetically. For the Hilbert curve, reading one byte of the parameter's mantissa will thus lead to four quaternary digits at once.
- Once the digits are computed, the recursive implementation can be changed into a simple loop of operator evaluations.
- The matrix parts of the operators will come down to only few different matrices, which correspond to the basic patterns of the space-filling curve (for the Hilbert curve, we will obtain only four different matrices). Hence, we only need to store an integer number as pointer to the correct matrix part for each operator, which drastically reduces the amount of memory required for the lookup table. In a similar way, the amount of storage required to store the translation vectors of the operators can be strongly reduced.

Speedups of 5–6 can be expected by combining loop unrolling with all these optimisations.

4.6.2 From Recursion Unrolling to State Diagrams

The recursion unrolling concept, as in Eq. (4.4) can be pushed further, if we write the Hilbert operators $H_0, \ldots, H_3$ in the following form:

$$H_0\colon \begin{pmatrix} x \\ y \end{pmatrix} \to \frac{1}{2}\left(T_0 \begin{pmatrix} x \\ y \end{pmatrix} + t_0\right) \text{ where } T_0 := \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}, \quad t_0 := \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

$$H_1\colon \begin{pmatrix} x \\ y \end{pmatrix} \to \frac{1}{2}\left(T_1 \begin{pmatrix} x \\ y \end{pmatrix} + t_1\right) \text{ where } T_1 := \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad t_1 := \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

$$H_2\colon \begin{pmatrix} x \\ y \end{pmatrix} \to \frac{1}{2}\left(T_2 \begin{pmatrix} x \\ y \end{pmatrix} + t_2\right) \text{ where } T_2 := \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad t_2 := \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

$$H_3\colon \begin{pmatrix} x \\ y \end{pmatrix} \to \frac{1}{2}\left(T_3 \begin{pmatrix} x \\ y \end{pmatrix} + t_3\right) \text{ where } T_3 := \begin{pmatrix} 0 & -1 \\ -1 & 0 \end{pmatrix}, \quad t_3 := \begin{pmatrix} 2 \\ 1 \end{pmatrix}$$

Performing successive unrolling steps, as in Eq. (4.4), we obtain the following arithmetisations for parameters $t = 0_4.q_1q_2q_3q_4$ or $\tilde{t} = 0_4.q_1q_2q_3q_4q_5q_6$:

$$\begin{aligned} h(t) = s(0_4.q_1q_2q_3q_4) &= \frac{1}{4}\left(T_{q_1q_2}\frac{1}{4}\left(T_{q_3q_4}\begin{pmatrix} 0 \\ 0 \end{pmatrix} + t_{q_3q_4}\right) + t_{q_1q_2}\right) \\ &= \frac{1}{4}\left(T_{q_1q_2}\frac{1}{4}t_{q_3q_4} + t_{q_1q_2}\right) = \frac{1}{4^2}T_{q_1q_2}t_{q_3q_4} + \frac{1}{4}t_{q_1q_2}, \end{aligned}$$

and similarly:

$$\begin{aligned} h(\tilde{t}) = h(0_4.q_1 \dots q_6) &= \frac{1}{4}\left(T_{q_1q_2} h(0_4.q_3 \dots q_6) + t_{q_1q_2}\right) \\ &= \frac{1}{4}\left(T_{q_1q_2}\left(\frac{1}{4^2} T_{q_3q_4} t_{q_5q_6} + \frac{1}{4} t_{q_3q_4}\right) + t_{q_1q_2}\right) \\ &= \frac{1}{4^3} T_{q_1q_2} T_{q_3q_4} t_{q_5q_6} + \frac{1}{4^2} T_{q_1q_2} t_{q_3q_4} + \frac{1}{4} t_{q_1q_2}. \end{aligned}$$

Repeating the computation for increasing numbers of digits, we obtain the following representation of the Hilbert mapping:

$$h(t) = h(0_4.q_1q_2 \dots q_{2n}) = \sum_{k=1}^{n-1} \frac{1}{4^k} T_{q_1q_2} \cdots T_{q_{2k-1}q_{2k}} t_{q_{2k+1}q_{2k+2}} + \frac{1}{4} t_{q_1q_2}.$$

Similar to $T_{q_1q_2} = T_{q_1} T_{q_2}$, we define $T_{q_1 \dots q_{2k}} := T_{q_1q_2} \cdots T_{q_{2k-1}q_{2k}}$, which leads to the following formula for the Hilbert mapping:

$$h(t) = h(0_4.q_1q_2 \dots q_{2n}) = \sum_{k=1}^{n-1} \frac{1}{4^k} T_{q_1 \dots q_{2k}} t_{q_{2k+1}q_{2k+2}} + \frac{1}{4} t_{q_1q_2}. \tag{4.5}$$

The operators $T_{q_1 \dots q_{2k}}$ can be successively computed as $T_{q_1 \dots q_{2k+2}} := T_{q_1 \dots q_{2k}} T_{q_{2k+1}q_{2k+2}}$ and will only take four different values (incl. those of T_0, T_1, and T_3), which correspond to the four basic patterns H, A, B, and C, as used in the grammar for the Hilbert curve. Function `hilbertBially` provides a non-recursive implementation of the resulting scheme to compute the Hilbert mapping. The function uses two for-loops. The first loop will compute the sequence of quaternary digits. The second for-loop will hold the (successively updated) value for $T_{q_1 \dots q_{2k}}$ in a variable `T`, and thus add the next term of the sum in (4.5) using tabulated values for the $t_{q_{2k-1}q_{2k}}$ (in `ttable[]`). The occurring instances of the operators $T_{q_{2k+1}q_{2k+2}}$ are stored in a table `Ttable`, which is used to update `T` in each iteration. Function `hilbertBially` can be further optimised by using a table that will return the new operator $T_{q_1 \dots q_{2k+2}}$ for all four values of $T_{q_1 \dots q_{2k}}$, depending on the digits q_{2k+1} and q_{2k+2}, and thus avoids an expensive operator evaluation in `T := T(Ttable[q[n]])`. Hence, compared to a recursion-based implementation, such an iterative implementation can, in addition, avoid operator executions.

Towards Finite State Machines

Due to the term $\frac{1}{4^k}$ in front of the term $T_{q_1 \dots q_{2k}} t_{q_{2k-1}q_{2k}}$, one could expect that the addition effectively appends digits to the current value of x in each step. This requires, however, that the elements of the added vector $T_{q_1 \dots q_{2k}} t_{q_{2k-1}q_{2k}}$ only take

Function hilbertBially – loop-based implementation of Hilbert mapping (using tables)

```
Function hilbertBially(t, depth)
    Parameter: t: index parameter, $t \in [0,1]$
               depth: depth of recursion (equiv. to number of quaternary digits)
begin
    for n = 1, ..., depth do
        // compute quaternary digits in array q
        q[n] := floor(4*t);
        t := 4*t - q[n]
    end
    // compute image point in x
    x := (0,0); T := $T_0$;
    for n = 1, ..., depth do
        // accumulate rotated translation vectors
        x := x + T( ttable[q[n]] ) / $4^n$;
        // accumulate rotation operators
        T := T(Ttable[q[n]]);
    end
    return x;
end
```

values in $0, \dots, 3$. Unfortunately, this is not the case for the canonical Hilbert operators. Respective operators for the Lebesgue curve (see Sect. 7.2) will lead to this property. An algorithm analogous to Function `hilbertBially` would then essentially turn into a finite state machine that takes the digits `q[n]` as input, and successively appends the digits of the image point to the output x. Such a construction to compute space-filling curves was first described by Bially [42] for the Hilbert curve – see also the following references section. To derive it via a construction similar to Eq. (4.5) requires a couple of technicalities. In particular, the operators should be constructed for a Hilbert curve on the square $[-1,1] \times [-1,1]$, and a final transformation is then necessary to map this curve back into the unit cube, $[0,1]^2$ – see Exercise 4.8.

References and Further Readings

For the arithmetisation, we followed the notation and presentation given in Sagan's textbook [233]. The general construction, however, is much older: Knopp, in 1917, already presented a uniform method to construct the Koch curve (see Sect. 5.3) and the Sierpinski space-filling curve (see Chap. 6) using the binary representation of parameters and mapping the respective digit sequences to sequences of nested triangles. Wunderlich [276], in 1954, extended this approach to general number systems on base r and sets of mappings $A_0, \dots A_{r-1}$, and gave an arithmetic representation for the Hilbert curve (and for several other continuous non-differentiable curves, including the Koch curve), which leads to formulas analogous to Eqs. (4.2) and (4.3). In a later, review-like article on space-filling curves [277], he presented arithmetisations of the Peano curves and of the Sierpinski curve. He also described the unrolling procedure given in Eq. (4.4) as a means to compute the Hilbert mapping for rational parameters t (where we "unroll" over the periodic digits – compare Exercise 4.1). Sagan [231] extended Wunderlich's arithmetic representation towards a closed formula for the Hilbert curve – the respective formula further simplifies Eq. (4.5) by studying the values of the operators $T_{q_1 \dots q_{2k}}$. Sagan provides an algorithm for the computation in [233].

As shown in Sect. 4.6.2, Sagan's unrolling of operators also leads to an algorithm that resembles a finite state machine. Bially's algorithm [42], one of the very first algorithms for space-filling curves, derived an equivalent algorithm directly via state machines. His approach was extended by Lawder and King [155], in particular for curves of higher dimension (however, Bially already provided state diagrams for a 3D and a 4D Hilbert curve).

A couple of further algorithms also build on the operator concept: Cole [69] introduced a recursive algorithm that implemented the rotation and translation operators directly via corresponding graphics primitives. Breinholt and Schierz [47] introduced a recursive algorithm ("Algorithm 781") to compute the iterations of the 2D Hilbert curve that is based on integer versions of the operators H_i and codes the current orientation of Hilbert iterations via two 0/1 parameters that reflect the classical grammar patterns. While the table-based implementation of space-filling mappings by Jin and Mellor-Crummey (SFCGen [141]) is primarily based on the grammar representation, they use the arithmetisation of curves to compute the respective tables. Further algorithms to compute space-filling curves were introduced by Butz [58, 59], in [198], or by Liu and Schrack [169] (2D Hilbert curve).

What's next?

- ➔ The next chapter will take a closer look at approximating polygons, which we used as helpers for construction, so far. They also provide an interesting connection to fractal curves, such as the well-known Koch curve.
- ↘ You may skip this chapter for the moment, and read on with Chap. 6, which deals with Sierpinski curves (constructed on triangular cells).
- ↱ At this time, we have all the necessary tools to leave the 2D world and move on to "proper", 3D space-filling curves – the respective constructions, grammars, and arithmetisation will be discussed in Chap. 8.

Exercises

4.1. The quaternary representation of a parameter t is defined via the equation

$$0_4.q_1q_2q_3q_4\cdots := \sum_{n=1}^{\infty} 4^{-n} q_n. \tag{4.6}$$

Compute the quaternary representation of the parameters $\frac{1}{8}$ and $\frac{1}{3}$, as well as of $\frac{2}{5}$.

4.2. Compute the image points of the Hilbert curve for the parameters $\frac{1}{3}$ and $\frac{2}{3}$, i.e. $h\left(\frac{1}{3}\right)$ and $h\left(\frac{2}{3}\right)$. In a similar way, compute $h\left(\frac{2}{5}\right)$.

4.3. Show that the point $\left(\frac{1}{2}, \frac{1}{2}\right)$ is the Hilbert image of three different parameters.

4.4. Give an arithmetisation of the Hilbert-Moore curve. Remember that the Hilbert-Moore curve consists of the connection of four suitably transformed Hilbert curves!

4.5. In analogy to the arithmetisation of the Hilbert and Peano curve, derive the transformation operators required to descibe the Peano-Meander curve, (as given in Fig. 2.8b on page 27), and give the formula to compute the values of the respective mapping $m(t)$.

4.6. For the recursion-unrolling arithmetisation of the Hilbert curve, as given in Eq. (4.4), computing the corresponding mappings $H_{q_i q_j}$. How many different rotation matrices occur in the operators?

4.7. Adopt the recursion-unrolling approach to derive more efficient algorithms to compute the inverse of the Hilbert mapping, i.e., the Hilbert index. In particular, derive an algorithm that corresponds to Function `hilbertBially`.

4.8. Derive the operators for a Hilbert mapping that maps the unit interval to the square $[-1, 1] \times [-1, 1]$ (instead of $[0, 1]^2$). Try to turn this mapping into a finite-state-machine representation, as suggested in Sect. 4.6.2.

Chapter 5
Approximating Polygons

5.1 Approximating Polygons of the Hilbert and Peano Curve

In Sect. 2.3.2 we have already used the approximating polygon of the Hilbert curve to guide us during the construction of the curve. The arithmetisation of the Hilbert curve, as introduced in the previous section, also gives us a tool to give a mathematical definition of the polygons:

Definition 5.1 (Approximating Polygon of the Hilbert Curve). The polygon that connects the $4^n + 1$ points

$$h(0), h(1 \cdot 4^{-n}), h(2 \cdot 4^{-n}), \ldots, h((4^n - 1) \cdot 4^{-n}), h(1),$$

is called the *n-th approximating polygon of the Hilbert curve*.

Figure 5.1 again plots the first three approximating polygons of the Hilbert curve. We used the approximating polygons to help during the construction of the Hilbert curve, which was also due to the following properties of the polygons:

- The approximating polygons of the Hilbert curve are defined on *corners* of the recursively substructured square. The respective corner points are the image points of the boundaries of the nested intervals at the same recursion level.
- The connected corner points are the start and end points of the Hilbert curve in the respective subsquares; hence, they determine where the curve enters and leaves a subsquare (i.e., the transfer into the next subsquare).

Note that we could define respective mappings $p_n(t)$ that describe the approximating polygons as curves, themselves. It can be shown that the respective sequence of mappings $p_n(t)$ *uniformly* converges to the Hilbert curve. With the obvious continuity of the polygons, this is a further proof for the continuity of the Hilbert curve.

M. Bader, *Space-Filling Curves*, Texts in Computational Science and Engineering 9,
DOI 10.1007/978-3-642-31046-1_5, © Springer-Verlag Berlin Heidelberg 2013

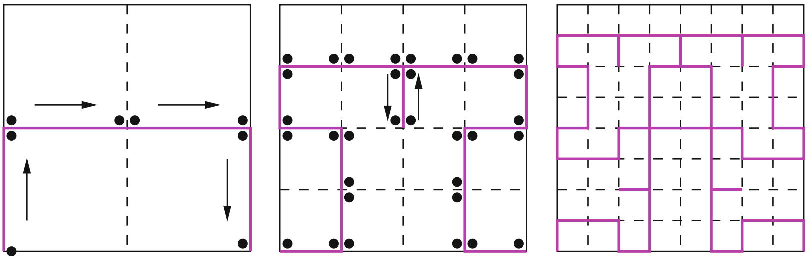


Fig. 5.1 The first three approximating polygons of the Hilbert curve

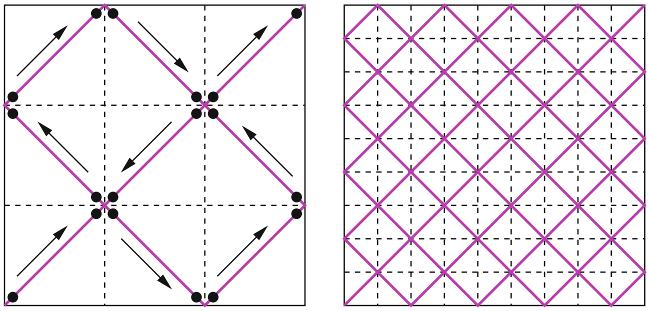


Fig. 5.2 First and second approximating polygon of the Peano curve

Approximating Polygons of the Peano Curve

Analogous to the Hilbert curve, we can define the approximating polygons for the Peano curve.

Definition 5.2 (Approximating Polygons of the Peano Curve). The polygon that connects the $9^n + 1$ points

$$p(0), p(1 \cdot 9^{-n}), p(2 \cdot 9^{-n}), \ldots, p((9^n - 1) \cdot 9^{-n}), p(1)$$

(p the Peano mapping) is called *n-the approximating polygon of the Peano curve*.

The first and second approximating polygons of the Peano curve are plotted in Fig. 5.2. Note the typical, diagonal-based structure, which results from the fact that entry and exit points of the Peano curve in each subsquare are always diagonally across.

Recursive Construction via a Generator

If we examine the successive construction of the approximating polygons of the Hilbert and the Peano curve, we recognise an important construction principle: the

$(n + 1)$-th approximating polygon is always generated by replacing each edge of the n-th polygon by a scaled-down first-order polygon. In addition to the scaling, we have to ensure the correct orientation of that basic polygon, in particular for the Hilbert curve. There, the polygon always needs to be oriented such that it is entirely within the respective subsquare. Hence, the approximating polygon is generated by recursive application of a basic polygon pattern, which we will call the *generator* in the following. The respective similarity to Koch curves and other fractal curves will be discussed in Sects. 5.3 and 5.4.

5.2 Measuring Curve Lengths with Approximating Polygons

We have defined the approximating polygons as the direct connection of the points

$$h(0), h(1 \cdot 4^{-n}), h(2 \cdot 4^{-n}), \ldots, h((4^n - 1) \cdot 4^{-n}), h(1).$$

In particular, all points connected by the polygon lie on the Hilbert curve. This is exactly the standard approach to measure the length of a given curve: we approximate the curve by a sequence of more or less equally distributed curve points, and sum up the distances between these points, i.e. measure the length of the polygon. For finer and finer resolution, i.e. if the distances between the measurement points are successively reduced, we expect that the respective measurement will become more and more accurate. The length of the curve is then defined as the limit of this procedure, if such a limit exists.

What Is the Length of the Hilbert Curve?

Hence, we can compute the length of the approximating polygons of the Hilbert curve, and use them to determine the length of the Hilbert curve. For the length of the polygons, we observe:

- The polygons are constructed by successive repetition of the generator. The generator consists of four units (up-right-right-down, e.g.) and replaces a polygon line that is only two units long.
- Hence, the length of the approximating polygons will **double** from each iteration to the next.

The length of the 0-th approximating polygon is 1 (the base line of the unit square); the length of the 1-st approximating polygon is 2, respectively. For the n-th polygon, we obtain a length of 2^n. Hence, for $n \to \infty$, the length grows infinitely, and our method to measure the length fails, because no limit of the polygon lengths exists. We conclude:

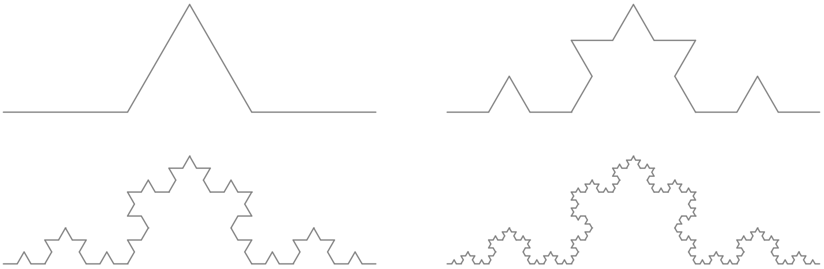


Fig. 5.3 The first four iterations of the Koch curve

1. The length of the Hilbert curve is obviously not well defined.
2. Instead, we can easily assign an area to the Hilbert curve. The area of the unit square, i.e. the area taken up by all points of the curve, is a natural definition for the area of the Hilbert curve, and is equal to 1.

In that sense, the Hilbert curve seems to be a two-dimensional object, which we wouldn't normally associate with a curve.

5.3 Fractal Curves and Their Length

Curves that are constructed via successive application of a generator are an important family within the so-called *fractal* curves. Hence, the Hilbert curve is a fractal curve in that sense. A further example of a fractal curve is the well-known *Koch* curve, as illustrated in Fig. 5.3. The generator of the Koch curve is obtained by removing the middle third of a given line, and replacing it by the two legs of an equilateral triangle placed on the removed third. By successively replacing each line segment by the generator from each iteration to the next, we obtain the iterations of the Koch curve, as in Fig. 5.3. The Koch curve itself, similar to the Hilbert curve, is defined as the limit curve of this construction.

We can try to measure the length of the Koch curve in the same way as for the Hilbert curve. In fact the measurement procedure can be generalised to any fractal curve that results from such a generator-based construction:

- The (infinite) fractal curve is approximated by the finite iterations, where we assume that the points interpolated by the polygons lie on the curve. This is guaranteed if the two endpoints of the generator stay on the endpoints of the replaced line segment. If the line segments of the polygon have length ε, then ε characterises the accuracy of the measurement. We can then denote the length of the polygon as $L(\varepsilon)$, and interpret it as an approximation of the length of the fractal curve, measured with accuracy ε.
- The successive repetition of the generator gives us a simple formula for $L(\varepsilon)$. If in each iteration, a line segment of length r (measured in line units) is replaced

by a generator polygon of length q, then the length of the iterations increases by a factor of $\frac{q}{r}$ in each step. At the same time, the accuracy is improved by a factor of $\frac{1}{r}$. Hence, we obtain a recurrence equation for the length:

$$L\left(\frac{\varepsilon}{r}\right) = \frac{q}{r} L(\varepsilon), \qquad L(1) := \lambda$$

- Applying this formula n times, we obtain for the length of the n-th iteration:

$$L\left(\frac{1}{r^n}\right) = \left(\frac{q}{r}\right)^n L(1) = \frac{q^n}{r^n}\lambda$$

Relative to the accuracy $\varepsilon := \frac{1}{r^n}$, we obtain that

$$L(\varepsilon) = \frac{q^n}{r^n}\lambda = \varepsilon q^n \lambda \qquad \text{where} \quad r^n = \frac{1}{\varepsilon} \quad \text{or} \quad n = -\log_r \varepsilon.$$

And after a short computation, we get:

$$L(\varepsilon) = \lambda \varepsilon q^{-\log_r \varepsilon} = \lambda \varepsilon \varepsilon^{-\log_r \varepsilon \cdot \log_\varepsilon q} = \lambda \varepsilon^{1-D}, \quad \text{with} \quad D = \frac{\log q}{\log r},$$

where we used that $\log_r \varepsilon \cdot \log_\varepsilon q = \frac{\log \varepsilon}{\log r} \cdot \frac{\log q}{\log \varepsilon} = \frac{\log q}{\log r}$.

Hence, approximating the length of generator-defined fractal curve leads to the following formula for the length:

$$L(\varepsilon) = \lambda \varepsilon^{1-D} \qquad \text{with} \quad D = \frac{\log q}{\log r}. \tag{5.1}$$

Note that a finite length is only obtained for the case $D = 1$, which means that $q = r$. Then, we replace a line segment of length r by a polygon of length $q = r$, which therefore has to be a straight line, as well. Hence, our fractal curve is just a straight line in that case, and measuring the length of this straight line turns out to be pretty boring.

Imagine, for comparison, the length measurement for a “well-behaved” curve, such as a parabola or a circle line. There, we know that the measured curve length $K(\varepsilon)$, for $\varepsilon \to 0$, will converge to a fixed value λ (the length of the curve). As the ratio between successive approximate curve lengths will approach the value 1, $K(\varepsilon)$ behaves according to the formula $K(\varepsilon) = \lambda\varepsilon^0$, at least in an asymptotic sense. Hence, we obtain a value of $D = 1$ in this case.

It seems that λ in Eq. (5.1) is actually a length, at least if $D = 1$. However, what is D? For a straight line, i.e. a 1D object, we obtained that $D = 1$. For “well-behaved” curves, which have a finite length and are therefore 1D objects, as well, we still get $D = 1$. However, for the Hilbert curve, with $r = 2$ and $q = 4$, and similar for the Peano curve, with $r = 3$ and $q = 9$, we obtain the values

$$D = \frac{\log 4}{\log 2} = 2 \qquad \text{and} \qquad D = \frac{\log 9}{\log 3} = 2.$$

Hence, D seems to be a dimension – being 2 for the 2D Hilbert and Peano curve. If we allow us a careful look to Chap. 8, where 3D space-filling curves will be discussed, we can determine the values $r = 2$ and $q = 8$ from Fig. 8.2, which plots the approximating polygon of a 3D Hilbert curve. Hence, $D = 3$.

Indeed, D can be interpreted as the "true" dimension of a fractal curve, and together with Eq. (5.1) leads to a sensible way to characterise the length and dimension of fractal curves:

- D is the *fractal dimension* of the curve.
- λ is the length respective to that dimension.

For both the Hilbert curve and the Peano curve, the fractal dimension $D = 2$ is consistent with the geometrical dimension of the target domain. Measuring the "length" of the two curves in any other dimension, apart from $D = 2$ leads to a value of either 0 or ∞. In particular, measuring the volume (length in 3D) of the Hilbert curve will give the value 0, and the classical 1D length is infinite.

5.4 A Quick Excursion on Fractal Curves

As depicted in Fig. 5.3, the generator of the Koch curve replaces a 3-unit segment by a 4-unit polygon in each step. Hence, the length measurement according to Eq. (5.1) leads to the following formula for the length of the Koch curve:

$$L(\varepsilon) = \varepsilon^{1-\frac{\log 4}{\log 3}} \approx \varepsilon^{1-1.26186}, \tag{5.2}$$

where ε is the length of the line segments in the respective iteration. Hence, as the dimension D of the Koch curve, we obtain

$$D = \frac{\log 4}{\log 3} \approx 1.26186.$$

Thus, the Koch curve is neither a proper (1D-)curve, nor space-filling, as the Hilbert or Peano curve. Measuring the length of the curve leads to an infinite length, while the area of the curve is 0 (compare Exercise 5.1).

This "fractal" dimension has led to the term *fractal curves* for curves similar to the Koch curve. Mandelbrot [174] has described fractal curves in detail, and suggested their use for the description of quasi-natural objects – see the Koch snowflake in Fig. 5.4. He also pointed out that the dimension D is equivalent to the *Hausdorff* dimension of the respective curves.

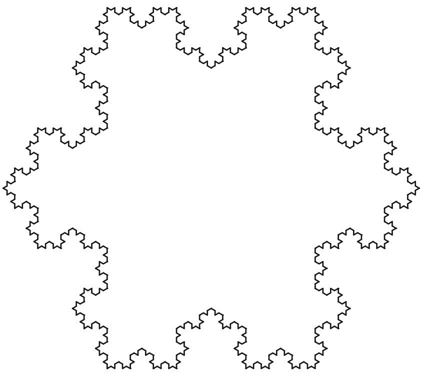


Fig. 5.4 The Koch snowflake (three Koch curves built on the legs of an equilateral triangle)

How Long Is the Coast Line of Britain?

Mandelbrot also presented the following, famous example for length measurements: Assume that we have to determine the length of the coast of the British Island. For a first, rough estimate we could, for example, take a respective map and a pair of dividers or a ruler. Measuring steps of 1 cm, we could then proceed step by step along the coast line, until we have an approximation of the entire coast. The number of centimetre-steps, together with the scale of the map, will give us a first estimate of the length of the coast line. In a second step, we could move to a collection of topographical maps with a much finer scale. The respective maps will also show more details – smaller bays will become visible, and we are likely to obtain a larger value for the length of the coast line. Figure 5.5 illustrates this effect by determining the coast length using two different resolutions. The measurements with the red lines uses units with only a third of the previous accuracy. The measured coast length thus increases from 108 to 149 units. Next, we could buy a surveying equipment, and physically perform a measurement of the coast lines, by determining the distances between characteristic points along the coast line. Such an expedition will probably lead to an even larger value for the coast length, and so on.

Luckily, this is a thought experiment, so we won't go on and consider measuring the coast line by an inch rule. From a modelling point of view, it is also debatable whether coast lines will show the same behaviour on all scales or along its entire extent. Sand beaches will probably lead to different fractal dimension than rocky cliffs, and on centimetre scales we might obtain different values for the fractal dimensions. However, the choice of resolution during the measurement obviously does have a significant influence on the result, and there seem to be certain length scales where coast lines – in their asymptotic behaviour – are more accurately modelled by fractal curves than by classical analytical curves. Mandelbrot supported his claim by the observation that different encyclopedias list different values for the lengths of coast lines and boundaries, which show discrepancies of up to 20 %. Moreover, he demonstrated that the measured coast lengths obey laws similar to those given in Eq. (5.1). Figure 5.6 illustrates this claim by comparing the respective

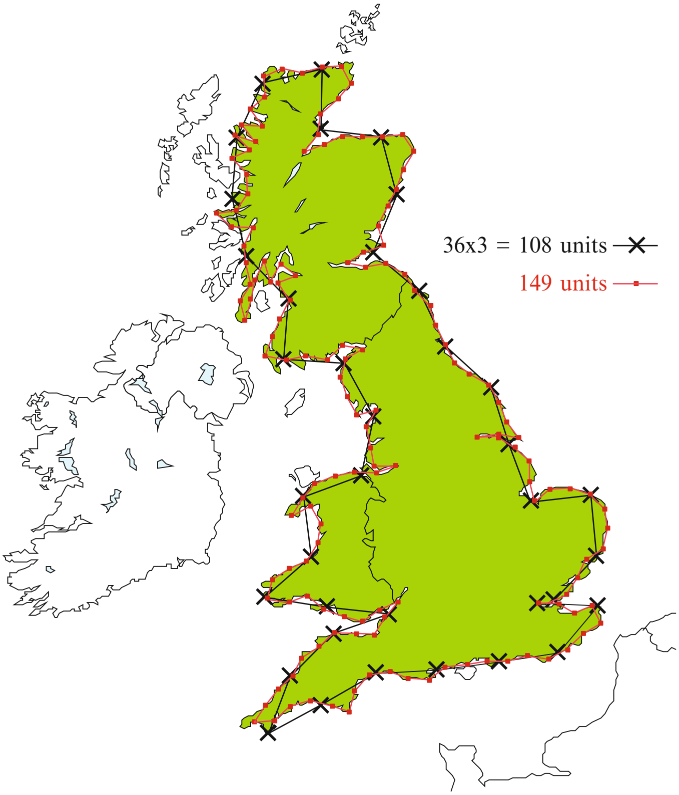


Fig. 5.5 Measuring the length of Britain's coast line – using two different approximations with different accuracy

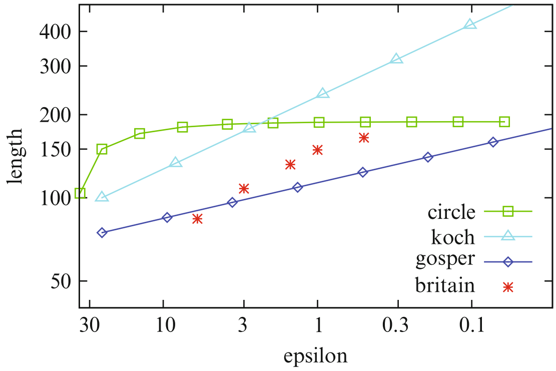


Fig. 5.6 Plotting the measured lengths of Britain's coast line, as determined from the map in Fig. 5.5, in comparison to the approximate lengths of the Koch curve, the Gosper island, and a circle

lengths of the British coast line (as estimated from the map in Fig. 5.5) with the lengths of iterations of the Koch curve and the Gosper island (see Sect. 7.5), and also versus approximations of a circle line.

References and Further Readings

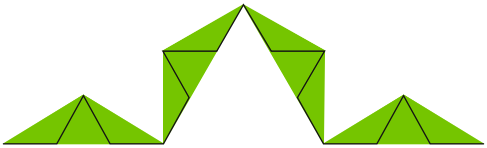
For the Hilbert and Peano curve, the iterations are much more common as an illustration than the approximating polygons. However, as a tool for construction, the polygons are important to determine entry and exit points. Wunderlich [277] pointed out that the respective connectivity conditions ("Anschlussbedingungen") are one of the central building blocks to construct space-filling curves. The term "approximating polygon", as defined in this chapter, was already used by Polya [220]. Constructions of space-filling curves that are based on approximating polygons are much more frequent for fractal-type curves – see, for example, the Gosper curve in Sect. 7.5. Ohno and Ohyama [199,200] presented a catalogue of symmetric and non-symmetric space-filling curves that have approximating polygons that do not touch themselves (or even intersect with themselves). For an introduction to fractal curves, and for more background on this topic, refer to Mandelbrot's book on "The Fractal Geometry of Nature" [174], but also to the respective chapter by Sagan [233], and to the review article by Goodchild and Mark [102], in particular due to the references given therein.

What's next?

- The next two chapters will introduce further space-filling curves. The Sierpinski curve (Chap. 6) is of special interest in Scienific Computing due to its construction on triangular cells.
- If you want to stick to the basics (Hilbert and Peano curves), you may skip the next two chapters, and go on with Chap. 8, which will deal with 3D curves.

Exercises

5.1. Using the highlighted triangular areas in the sketch below as upper bounds, compute the area of the Koch curve.



5.2. Using the highlighted triangles in the sketch of Exercise 5.1, we can generate an arithmetisation of the Koch curve in the usual way (mapping subintervals to subdomains). Set up that arithmetisation (note that the respective construction was already presented in 1917, by Knopp [150]).

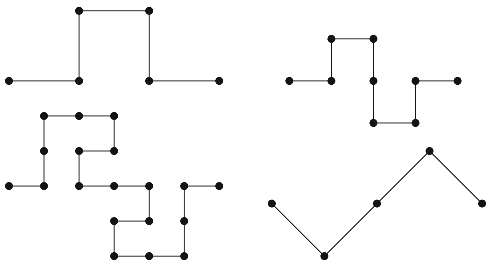
5.3. To construct the Koch curve, we replace the "middle third" by two legs of a triangle, where both legs have the lengths of the left and right "thirds". What happens, if we – instead of the middle third – take out a much smaller section (or even an infinitesimally small section)? This question was studied by Cesaro, in 1905 [64], and the resulting curves are also known as Cesaro curves.

5.4. Give a grammar representation for the iterations of the Koch curve. Try a "turtle" grammar, first, but also formulate a "plotter" grammar.

5.5. Give grammar representations for the approximating polygons of the Hilbert curve – again, try both a "turtle" grammar and a "plotter" grammar. Is it possible to generate the Hilbert polygons via the generator-based approach used for fractal curves?

5.6. The approximating polygons of the standard Peano curve cannot be represented as a fractal curve with uniform generator. However, there actually exists a Peano curve that is based on a single generator. Try to construct this curve.

5.7. Examine the generators given below (all curves are taken from [174]) and construct the first iterations of the resulting fractal curves, or write a program to generate the iterations. What's the fractal dimension of each curve?



Chapter 6
Sierpinski Curves

6.1 The Sierpinski-Knopp Curve

All space-filling curves discussed so far were based on a recursive substructuring into squares. The Sierpinski curve, in contrast, may be geometrically constructed using a recursive substructuring based on triangles. The curve is named after Waclaw Sierpinski, who – in 1912 – presented the respective mapping as the solution of certain functional equations. Similar to the Hilbert and Peano curve, the Sierpinski curve maps the unit interval onto a square. However, the original construction refers to the square $[-1,1]^2$ instead of the unit square. As can be seen from its illustration in Fig. 6.1, the curve may be split into two curves that each fill a right triangle, if the filled square is cut along the main diagonal. The respective triangle-filling curves will be discussed in the following section. A respective mathematical description was introduced by Konrad Knopp, which is why the respective curve is also called the Sierpinski-Knopp curve [233]. We will use the term Sierpinski-Knopp curve, only if we want to explicitly discriminate between the triangle- and square-filling version.

6.1.1 Construction of the Sierpinski Curve

The Sierpinski curve results from a recursive substructuring of a right, isosceles triangle. The original triangle is successively split into congruent subtriangles. Each triangle is split into two by splitting the hypotenuse. Hence, the former legs of the parent triangle form the hypotenuses of the two child triangles. The first six steps of this construction scheme are given in Fig. 6.2. Similar to the Hilbert and Peano curve, we obtain the self-similar iterations typical for space-filling curves. Following the usual procedure, we can define the Sierpinski-Knopp curve via nested intervals and the corresponding sequentialisations.

M. Bader, *Space-Filling Curves*, Texts in Computational Science and Engineering 9,
DOI 10.1007/978-3-642-31046-1_6, © Springer-Verlag Berlin Heidelberg 2013

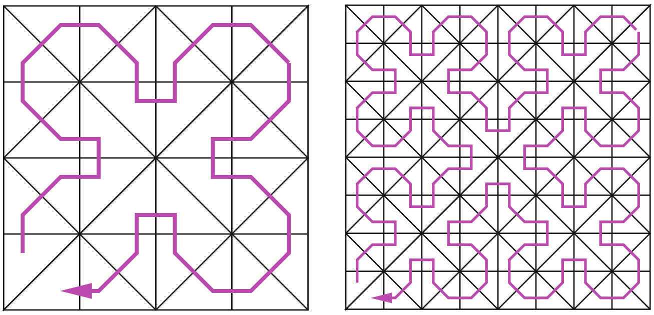


Fig. 6.1 Iterations of the Sierpinski curve filling a square

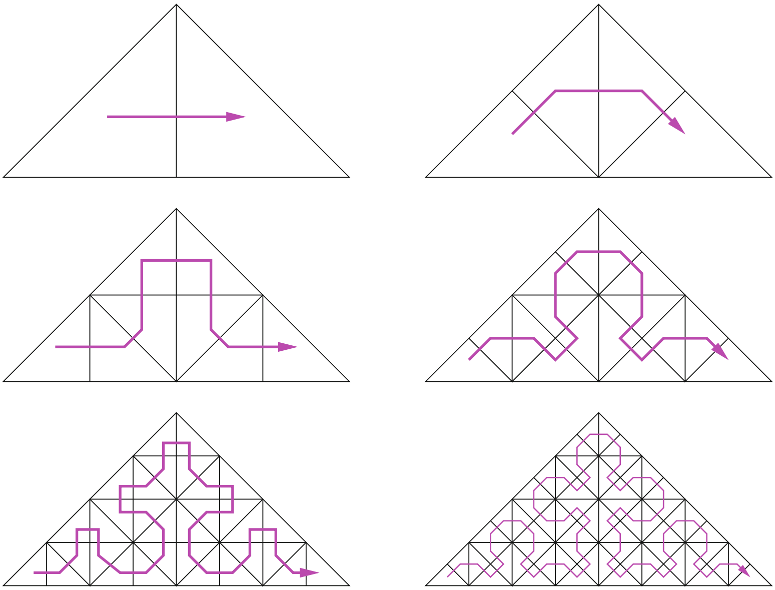


Fig. 6.2 The first construction steps of the Sierpinski-Knopp curve

Definition 6.1 (Sierpinski-Knopp Curve). Consider $\mathcal{I} := [0,1]$ and the triangle $\mathcal{T}$ defined by the corners $(0,0)$, $(1,1)$, and $(2,0)$. Then, define the function $s\colon \mathcal{I} \to \mathcal{T}$ via the following description:

- For each $t \in \mathcal{I}$, there is a sequence of nested intervals

$$\mathcal{I} \supset [a_1, b_1] \supset \ldots \supset [a_n, b_n] \supset \ldots,$$

  where each interval of the sequence results from a splitting of the parent interval into two equal parts: $[a_k, b_k] = [i_k \cdot 2^{-k}, (i_k + 1)2^{-k}], i_k = 0, 1, 2, \ldots, 2^k - 1$.

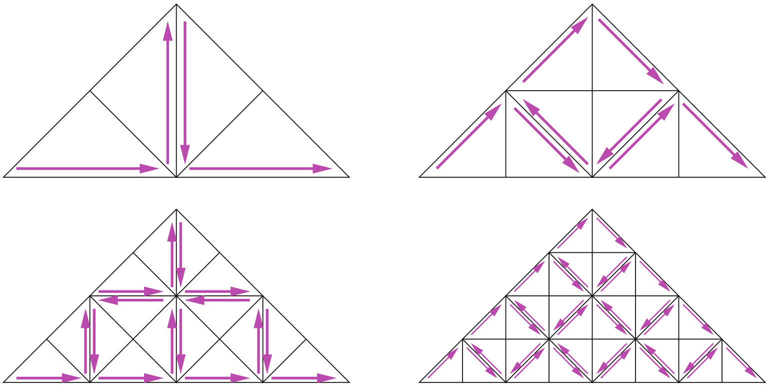


Fig. 6.3 Construction of the Sierpinski-Knopp curve using the approximating polygon

- Any such sequence of nested intervals corresponds to a sequence of nested, isosceles, right triangles. The substructuring of triangles, as well as the correspondence between intervals and triangles shall be given as shown in Fig. 6.2.
- The resulting sequence of nested triangles converges to a uniquely defined point in $\mathcal{T}$ – define this point as $s(t)$.

The image of the resulting mapping $s : \mathcal{I} \rightarrow \mathcal{T}$ is a space-filling curve, the *Sierpinski-Knopp curve*.

The proof of continuity works in exactly the same way as for the Hilbert curve. We exploit the recursive construction via nested intervals and triangles, and use that two triangles which are direct neighbours in the Hilbert order will share a common edge. The proof of surjectivity is also nearly identical to that for the Hilbert curve.

The Sierpinski-Knopp curve enters and leaves the subtriangles always in the corners adjacent to the hypotenuse of the triangle. Hence, the approximating polygon of the curve runs along the hypotenuse. This is reflected in Fig. 6.3.

6.1.2 Grammar-Based Description of the Sierpinski Curve

In Fig. 6.3, we see that the approximating polygons of the Sierpinski curve have an alternating form in even and odd iterations: for odd iterations, the polygons run in horizontal and vertical direction; for even iterations, they run along the diagonals. A straightforward grammar-based description of the Sierpinski curve would therefore require eight non-terminals – four non-terminals representing the horizontal and vertical patterns of the odd iterations, and four non-terminals representing the diagonal patterns.

We obtain a simpler grammar, if we simply leave out the even iterations, and use a substructuring that splits each triangular cell into four subtriangles in each

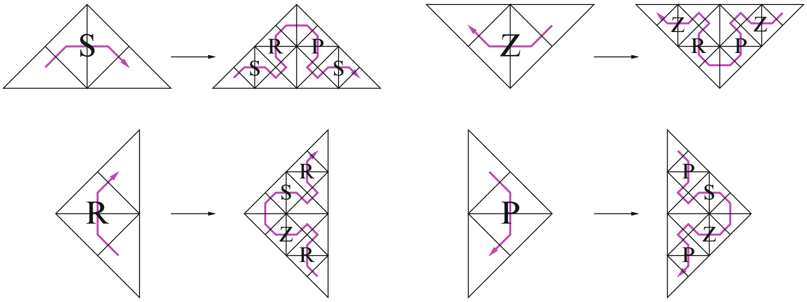


Fig. 6.4 Illustration of the basic patterns of the Sierpinski curve and of the production rules of the respective grammar

iteration steps. From the approximating polygons (compare the two images on the left of Fig. 6.3), we can set up a grammar that requires only four non-terminals. The terminals, non-terminals, and productions can be read from Fig. 6.4, which again illustrates the basic patterns and replacement rules of the iterations of the Sierpinski-Knopp curve. Using the terminal characters $\{\nearrow, \nwarrow, \searrow, \swarrow, \rightarrow, \uparrow, \downarrow, \leftarrow\}$, which now allow for diagonal moves, as well, we obtain the following productions of the grammar:

$$
\begin{aligned}
S &\longleftarrow S \nearrow R \rightarrow P \searrow S \\
R &\longleftarrow R \nwarrow Z \uparrow S \nearrow R \\
P &\longleftarrow P \searrow S \downarrow Z \swarrow P \\
Z &\longleftarrow Z \swarrow P \leftarrow R \nwarrow Z
\end{aligned}
$$

Again, we can use the grammar for a more formal description of the construction of the Sierpinski curve, and thus supplement Definition 6.1.

6.1.3 Arithmetisation of the Sierpinski Curve

As for the Hilbert and Peano curves, the arithmetisation of the Sierpinski curve works on a suitable number system. Similar to the grammar-based description, also the arithmetisation of the Sierpinski curve is simpler, if it is based on a substructuring of intervals and triangles into four parts in each iteration (see, in contrast, Exercise 6.1). Hence, analogous to the Hilbert curve, we start with the quaternary representation of the parameter t:

$$t = 0_4.q_1q_2q_3q_4\ldots$$

The coordinate function $s(t)$ of the Sierpinski-Knopp curve then has the representation:

$$s(0_4.q_1q_2q_3q_4\ldots) = S_{q_1} \circ S_{q_2} \circ S_{q_3} \circ S_{q_4} \circ \cdots \begin{pmatrix} 0 \\ 0 \end{pmatrix}. \tag{6.1}$$

As usual, the operators S_0 to S_3 have to be determined from the required rotation, reflections, and translations of the basic pattern. If we assume that the basic triangle to be filled is defined by the corners $(0, 0)$, $(1, 1)$, and $(2, 0)$, then we obtain:

$$S_0 := \begin{pmatrix} x \\ y \end{pmatrix} \to \begin{pmatrix} \frac{1}{2}x \\ \frac{1}{2}y \end{pmatrix} \qquad S_1 := \begin{pmatrix} x \\ y \end{pmatrix} \to \begin{pmatrix} -\frac{1}{2}y + 1 \\ \frac{1}{2}x \end{pmatrix}$$

$$S_2 := \begin{pmatrix} x \\ y \end{pmatrix} \to \begin{pmatrix} \frac{1}{2}y + 1 \\ -\frac{1}{2}x + 1 \end{pmatrix} S_3 \begin{pmatrix} x \\ y \end{pmatrix} \to \begin{pmatrix} \frac{1}{2}x + 1 \\ \frac{1}{2}y \end{pmatrix} \tag{6.2}$$

After the examples from Chap. 4, we can keep the explanation of the operators short. For S_0 and S_3, a simple scaling is sufficient, plus an additional translation for S_3. The transformation matrix for operator S_1 corresponds to a rotation by 90° (counterclockwise). Operator S_2 is obtained from S_1 via a rotation by 180°, such that an additional change of sign for both x- and y-direction is needed.

6.1.4 Computation of the Sierpinski Mapping

Analogous to the Function `hilbertBially` introduced in Sect. 4.6.2, we can derive an efficient algorithm to compute the Sierpinski mapping, which uses recursion unrolling to reduce the number of operations. Certain similarities between the operators $S_0, \ldots, S_3$, as given in Eq. (6.2), allow us to simplify the computation of the accumulated operators. Again, we rewrite the operators $S_0, \ldots, S_3$ in the following form:

$$S_0\colon \begin{pmatrix} x \\ y \end{pmatrix} \to \frac{1}{2}\left(Z_0 \begin{pmatrix} x \\ y \end{pmatrix} + z_0\right) \text{ where } Z_0 = Z_{p(0)} := \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad z_0 := \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

$$S_1\colon \begin{pmatrix} x \\ y \end{pmatrix} \to \frac{1}{2}\left(Z_1 \begin{pmatrix} x \\ y \end{pmatrix} + z_1\right) \text{ where } Z_1 = Z_{p(1)} := \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix}, \quad z_1 := \begin{pmatrix} 2 \\ 0 \end{pmatrix}$$

$$S_2\colon \begin{pmatrix} x \\ y \end{pmatrix} \to \frac{1}{2}\left(Z_3 \begin{pmatrix} x \\ y \end{pmatrix} + z_2\right) \text{ where } Z_3 = Z_{p(2)} := \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}, \quad z_2 := \begin{pmatrix} 2 \\ 2 \end{pmatrix}$$

$$S_3\colon \begin{pmatrix} x \\ y \end{pmatrix} \to \frac{1}{2}\left(Z_0 \begin{pmatrix} x \\ y \end{pmatrix} + z_3\right) \text{ where } Z_0 = Z_{p(3)} := \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \quad z_3 := \begin{pmatrix} 2 \\ 0 \end{pmatrix}$$

Here, the Z operators are chosen to represent rotations in 90°-steps, Z_1 being a rotation by 90° and Z_3 by 270°. Note that $Z_1 = -Z_3$ and

$$Z_1^2 = \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} 0 & -1 \\ 1 & 0 \end{pmatrix} = \begin{pmatrix} -1 & 0 \\ 0 & -1 \end{pmatrix} = -I =: Z_2,$$

hence, Z_2 shall be a rotation by 180°. The mapping $p(q)$ thus reflects how many 90°-steps are taken in the operator S_q for a digit q.

Following the procedure discussed in Sect. 4.6.2, we obtain the following representation of the Sierpinski mapping:

$$s(t) = s(0_4.q_1q_2\dots q_{2n}) = \sum_{k=1}^{n-1} \frac{1}{4^k} Z_{p(q_1q_2)} \cdots Z_{p(q_{2k-1}q_{2k})} z_{q_{2k+1}q_{2k+2}} + \frac{1}{4} z_{q_1q_2}.$$

With the definition $Z_{p(q_1\dots q_{2k})} := Z_{p(q_1q_2)} \cdots Z_{p(q_{2k-1}q_{2k})}$, we obtain the following formula for the Sierpinski mapping:

$$s(t) = s(0_4.q_1q_2\dots q_{2n}) = \sum_{k=1}^{n-1} \frac{1}{4^k} Z_{p(q_1\dots q_{2k})} z_{q_{2k+1}q_{2k+2}} + \frac{1}{4} z_{q_1q_2}. \tag{6.3}$$

In an implementation to compute $s(t)$ according to Eq. (6.3), we can again use a table to store the possible values of the translations $z_{q_iq_j}$ (16 vectors, if we combine two quaternary digits in each unrolling step). To obtain the operators $Z_{p(q_1\dots q_{2k})}$, we need to accumulate the respective 90°-rotations – due to our previous definitions, we obtain $p(q_1\dots q_k) = \sum q_j \mod 4$.

Function `sierpUnroll` provides a non-recursive implementation of the resulting scheme to compute the Sierpinski mapping, similar to Function `hilbertBially`. We can again speed up this algorithm, if we use respective tables to obtain the operators `Ztable[p[n-1]]`.

6.2 Generalised Sierpinski Curves

The construction principle of the Sierpinski curve can be easily generalised from isosceles, right triangles to arbitrary triangles. It is even possible to replace the edges by more complicated curves as boundaries. The respective modifications of the Sierpinski construction will be presented in the following section. They will also provide us with a third approach to compute iterations, approximating polygons, and mappings of space-filling curves, in general.

Function sierpUnroll – computing the Sierpinski mapping (with unrolling)

```
Function sierpUnroll(t, depth)
    Parameter: t: index parameter, $t \in [0,1]$
               depth: depth of recursion (equiv. to number of hex digits)
begin
    for n = 1, ..., depth do
        // compute hex digits in array q
        q[n] := floor(16*t);
        t := 16*t - q[n];
        // accumulate rotations in array p
        p[n] := ( p[n-1] + Ztable[q[n]] ) mod 4
    end
    // compute image point in x
    x := (0,0);
    for n = depth, ..., 1 do
        // accumulate rotated translation vectors
        x := x + Ztable[p[n-1]] ( ztable[q[n]] ) / $4^n$;
        return x;
    end
end
```

6.2.1 Bisecting Triangles Along Tagged Edges

In the classical geometrical construction of Sierpinski curves, all cells are right, isosceles triangles. The right angle and the hypotenuse define for each triangle how it has to be bisected for the next iteration. In each step the triangle is split along the hypotenuse. To generalise this construction for arbitrary triangles, we require a substitute for the hypotenuses. For that, we will use the concept of *tagged edges*.

Definition 6.2 (Triangles with Tagged Edges). Given are three points $x_1, x_2, x_3 \in \mathbb{R}^2$ that define the corners of a triangle. The triple

$$[x_1, x_2, x_3]$$

shall define a *triangle with tagged edge* x_1x_2. The tagged edge x_1x_2 is assumed to be oriented.

Analogous to the original Sierpinski construction, we can now recursively subdivide an initial triangle domain recursively along tagged edges. If x_4 is the splitting point on the tagged edge x_1x_2, then we will subdivide the tagged triangle x_1, x_2, x_3 into the two subtriangles

$$[x_1, x_3, x_4] \quad \text{and} \quad [x_3, x_2, x_4]\,.$$

Figure 6.5 illustrates this splitting, and also shows that the polygon $x_1x_3x_2$ along the new tagged edges connects the three corners x_1, x_3, and x_2 along the oriented,

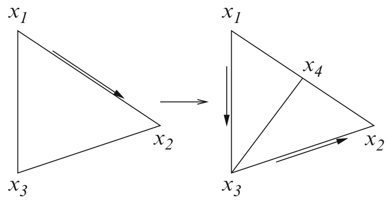


Fig. 6.5 Bisection of a triangle at the tagged edge. The oriented, tagged edges are highlighted by *arrows*

tagged edges. We know this construction principles from the approximating polygons of the Sierpinski curve. Hence, the tagged edges will build the approximating polygons of the generalised Sierpinski curve – as the hypotenuses did for the classical construction. In fact, this construction is already sufficient to define the generalised Sierpinski curve:

Definition 6.3 (Generalised Sierpinski Curve). Given are $\mathcal{I} := [0,1]$ and a target triangle $\mathcal{T} = [x_1, x_2, x_3]$ with tagged edge $x_1 x_2$. The function $\hat{s}\colon \mathcal{I} \to \mathcal{T}$ shall be defined via the following construction:

- For each parameter $t \in \mathcal{I}$, there is a sequence of nested intervals
$$\mathcal{I} \supset [a_1, b_1] \supset \ldots \supset [a_n, b_n] \supset \ldots,$$
where each interval in the sequence results from a bisection of its predecessor: $[a_k, b_k] = [i_k \cdot 2^{-k}, (i_k + 1)2^{-k}], i_k = 0, 1, 2, \ldots, 2^k - 1$.
- Each sequence of intervals corresponds to a uniquely defined sequence of nested triangles with tagged edges. Each triangle $[x_\alpha, x_\beta, x_\gamma]$ is bisected into the subtriangles $[x_\alpha, x_\gamma, x_\delta]$ and $[x_\gamma, x_\beta, x_\delta]$ (x_δ a point on the tagged edge x_α, x_β). The initial triangle of the sequence is given by T.
- The constructed sequence of triangular domains converges to a uniquely defined point in $\mathcal{T}$ – this point shall be defined as $\hat{s}(t)$.

The image of the function $\hat{s} : \mathcal{I} \to \mathcal{T}$ is a space-filling curve, and shall be called a *generalised Sierpinski curve*.

Note that we did not determine the choice of the splitting point x_δ on the tagged edge. A straightforward choice would be to take the midpoint of the tagged edge, but other choices are possible, as well. Figure 6.6 shows a generalised Sierpinski curve on an equilateral triangle (with mid-point splitting). The respective curve was constructed by Algorithm 6.1, which provides a straightforward implementation of the construction outlined in the definition. The algorithm generates the list of centre points of all triangles on a given bisection level n – the polygon defined by these centre points, as usual, is the n-th iteration of the curve. Note that the sequence and relative geometrical orientation of the three parameters `x1`, `x2`, and `x3` determine the local pattern of the generalised Sierpinski curve. Hence, a respective algorithm can also be derived for the Hilbert or Peano curves – see the paragraph on *vertex-labelling algorithms* on page 89 and also the exercises for this section.

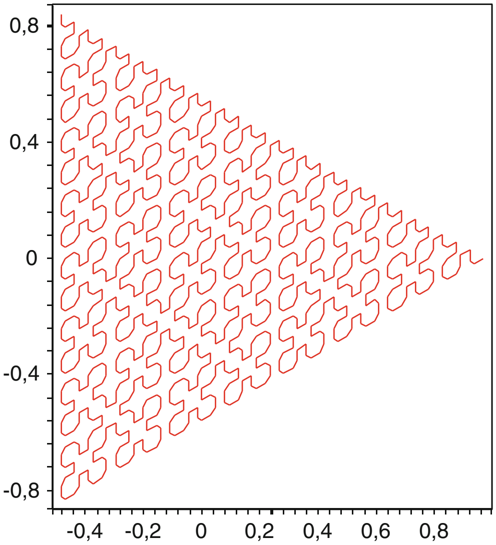


Fig. 6.6 Generalised Sierpinski curve on an equilateral triangle (10-th iteration)

Algorithm 6.1: Computing the n-th iteration of a generalised Sierpinski curve

```
Procedure gensierp(x1, x2, x3, n)
    Parameter: x1,x2,x3: vertices (as 2D coordinates); n: refinement level
    Data: curve: list of vertices (empty at start, contains iteration points on exit)
begin
    if n = 0 then
        // add center point of triangle x1,x2,x3 to output list:
        return attach(curve, center(x1,x2,x3))
    else
        // compute bisection vertex:
        xnew := mid(x1,x2);
        // recursive call to child triangles:
        gensierp(x1, x3, xnew, n-1);
        gensierp(x3, x2, xnew, n-1);
    end
end
```

6.2.2 *Continuity and Locality of Generalised Sierpinski Curves*

For equilateral, right start triangles, the generalised Sierpinski curve is identical to the "classical" Sierpinski curve. Thus, the generalised Sierpinski curves follow the same construction, and should also be representable by grammars, or by respective arithmetisations. We will therefore at least shortly discuss such representations for the generalised Sierpinski curves.

Figure 6.7 shows the well-known grammar-based construction scheme for the (generalised) Sierpinksi curve, based on four patterns S, R, P, and Z. From the

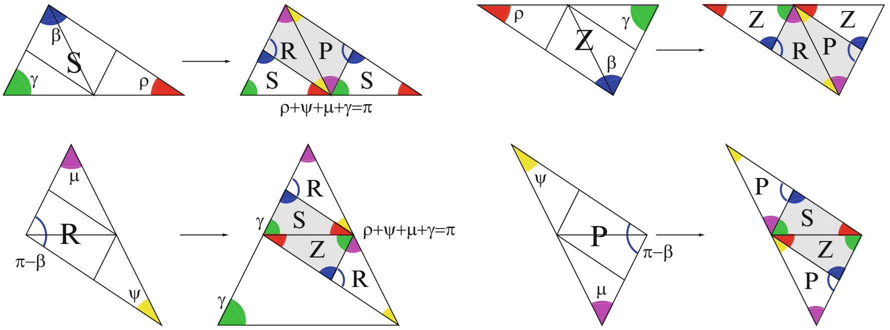


Fig. 6.7 Geometric patterns during the construction of a generalised Sierpinski curve. The analysis of the *highlighted angles* proves that the subtriangles may be classified into exactly four congruency classes

construction, we quickly suspect that all triangles that belong to the same pattern are congruent, i.e. can be transformed into each other by simple scaling and translation operations. With a careful examination of the angles of the triangles, as illustrated in Fig. 6.7, we can also prove this:

- An S triangle is substructured into two S triangles, an R and a P triangle. To prove that the smaller S triangles are scaled down variants of the original S triangle is straightforward (theorem of intersecting lines). The two triangles R and P form a parallelogram, and thus have the same angles (and are therefore congruent to each other).
- The R triangle will again be substructured, which will lead to two R triangles (again congruent due to the theorem of intersecting lines), a new pattern Z, and a triangle that should be congruent to the S triangles. S and Z form a parallelogram, and therefore have the same angles.
- From the illustration of the R-production in Fig. 6.7, we see that the lower-left angle of S is equal to γ. Together with the angle β, the smaller S triangle is proven to be congruent to the original S triangle.
- From triangle P, we read that $\psi + \mu + (\pi - \beta) = \pi$, hence $\psi + \mu = \beta$. In addition, we have that $\rho + \gamma = \pi - \beta$ (from triangle S). The two parallelograms built by either R plus P or S plus Z are therefore congruent. Splitting them along either of the diagonals leads to the patterns R plus P or S plus Z, respectively. With that knowledge, we can quickly identify all angles, as illustrated in Fig. 6.7.

Hence, during the construction of the generalised Sierpinski curve, we obtain only four different shapes for the subtriangles, which are given by the four patterns S, R, P, and Z. All triangles with the same patterns can be mapped onto each other by scaling and translation operations, only. Moreover, all S and Z triangles are congruent; the Z triangles are rotated by 180°. Likewise, the R and P triangles are congruent to each other.

The fact that all triangles in the generalised Sierpinski construction fall into only a few congruency classes has a couple of important consequences:

- For applications in scientific computing (finite element methods, for example), the size of the smallest and largest angle of the triangle cells is a limiting factor for accuracy. The congruency classes guarantee that we can determine these two angles easily, and that neither minimum nor maximum angle will approach any extreme values.
- In Chap. 11, we will see that the congruency of the subtriangles is an important prerequisite for certain locality properties of space-filling curves.
- In particular, our established proof for continuity of a space-filling curve will work for generalised Sierpinski curves, as well: as the triangles are all scaled-down copies of four basic templates, we can derive the required relation between size of nested intervals and respective diameter of triangles.

In Sects. 8.3 and 12.2, we will discuss three-dimensional variants of the Sierpinski curve. There, we will see that substructuring rules that do not lead to subtriangles from a few congruency classes lead to problems in the construction of proper space-filling curves.

6.2.3 Filling Triangles with Curved Edges

In Definition 6.3, we required that the new corner point x_δ is chosen somewhere on the tagged edge. This choice gives us two important guarantees:

1. In every iteration, the boundary defined by the union of all subtriangles will stay identical to the boundary of the initial triangle $\mathcal{T}$. This is because any tagged edge on a boundary of $\mathcal{T}$ will be split in two edges that are still placed on the boundary of $\mathcal{T}$. We will actually change this aspect in the following subsection.
2. Due to the construction, a tagged edge will always be the tagged edge for two adjacent triangles. If both triangles are bisected, the four subtriangles should neither have an overlap (apart from the edge itself) or a gap between them. This is always guaranteed, if we choose the same splitting point for both triangles, but it is also sufficient to choose splitting points on the tagged edge.

If we want to define a Sierpinski curve for a target domain with a more complicated boundary, we need to disregard the first guarantee. Hence, we will allow that, during the bisection

$$[x_1, x_2, x_3] \to [x_1, x_3, x_4], [x_3, x_2, x_4],$$

the new splitting point x_4 is no longer chosen on the edge x_1x_2. As illustrated in Fig. 6.8, we may move it onto the edge of the target domain, instead. We interpret this construction such that a domain given by the nodes $[x_1, x_2, x_3]$ is no longer necessarily a triangle. Instead, any “edge” x_1x_2 may be given by a more complicated curve. In practice, it will make sense to retain straight lines as edges in the interior

Fig. 6.8 Modified bisection of a triangle with tagged edge, such that the new corner is placed on a complicated boundary

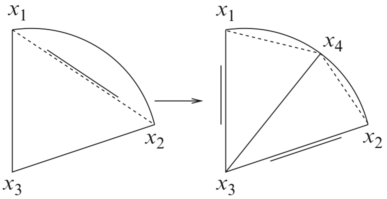
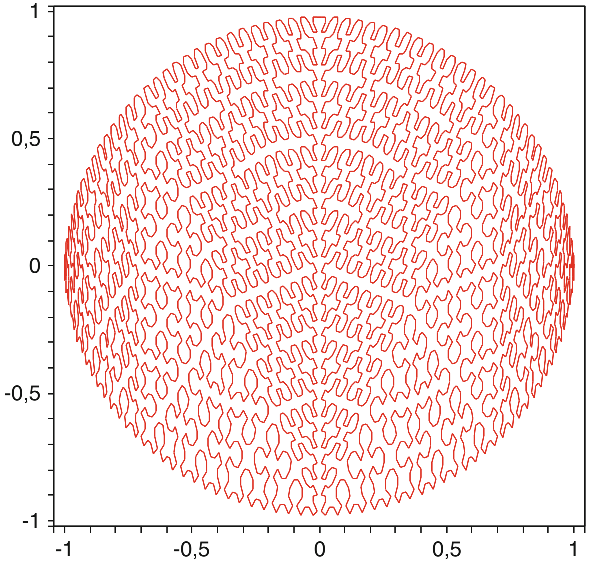


Fig. 6.9 A circle-filling generalised Sierpinski curve



of the target domain, and only allow free-form curves along the boundaries of $\mathcal{T}$. An alternate interpretation is that we only consider regular triangles as cells, but obtain a polygonal approximation of the complicated boundary by moving triangle corners, as in Fig. 6.8.

A possible implementation is to extend Algorithm 6.1 to provide a parameter representation of a curve for each edge of the initial target domain. The splitting points are then computed from this parameter representation, and we can generate a generalised circle-filling Sierpinski curve, as given in Fig. 6.9.

References and Further Readings

Already in 1905, Cesaro [64] essentially described the construction of the triangle-filling Sierpinski curve as a generalisation of the Koch curve (compare Exercise 5.3). Sierpinski curve introduced “his” curve in 1912 [244] based on a functional equation, and only briefly mentioned its geometrical construction. Such a construction, as triangle-filling curve, was discussed by Polya, one year later [220], who generalised the curve to right triangles. The standard approach to construct the mapping via subdivision of intervals and triangles was described by Knopp in 1917 [150]. In Scientific Computing, the triangular refinement scheme adopted for the Sierpinski-curve construction, including the construction of generalised curves, is equivalent to the so-called *newest vertex bisection* (e.g., [187, 242]). Sierpinski orders on such grids often exploit the respective refinement structure – see especially Chap. 9.

Vertex-Labeling Algorithms

In this chapter, Algorithm 6.1 for the generalised Sierpinski curves was derived from the representation of triangular cells (and the respective bisection rules) via vertex triples. The technique, however, can be easily adapted to describe Hilbert, Peano, and other space-filling curves in two and three dimensions. Bartholdi and Goldsman introduced such algorithms as *vertex-labelling algorithms* [29]. Algorithm 6.2 is a respective implementation of a Hilbert traversal. Properly modified, vertex-labelling algorithms can be used to compute the Hilbert mapping and Hilbert index, as well. Note that the order of parameters $x1, x2, x3, x4$ encodes the geometrical orientation of the basic patterns: $x1$ and $x4$ determine entry and exit point in each square, while $x2$ and $x3$ denote the remaining two corners in the sequence as they are visited by the Hilbert curve.

Algorithm 6.2: Computing the n-th iteration of a Hilbert curve via vertex-labelling

```
Procedure hilbertVL(x1, x2, x3, x4, n)
    Parameter: x1,x2,x3,x4: vertices (as 2D coordinates); n: refinement level
    Data: curve: list of vertices (empty at start, contains iteration points on exit)
begin
    if n > 0 then
        return attach(curve, center(x1,x2,x3,x4))
    else
        hilbertVL(x1, mid(x1,x4), mid(x1,x3), mid(x1,x2), n-1);
        hilbertVL(mid(x1,x2), x2, mid(x2,x3), mid(x2,x4), n-1);
        hilbertVL(mid(x1,x3), mid(x2,x3), x3, mid(x3,x4), n-1);
        hilbertVL(mid(x3,x4), mid(x2,x4), mid(x1,x4), x4, n-1);
    end
end
```

For three- and higher-dimensional Hilbert and Peano curves, vertex-labelling algorithms become less handy, because of the exponentially increasing number of parameters – a d-dimensional (hyper-)cube will have 2^d vertices, which leads to

2^d parameters for a vertex-labelling algorithm. There is, of course, a substantial amount of redundancy in the vertex-tuple coding, but reducing that redundancy by just storing the orientation of the cube according to the basic pattern, for example, will lead to algorithmic approaches that are based on arithmetisation (compare the algorithm by Breinholt and Schierz [47], for example, as discussed on page 63). For simplex-based construction, such as (three-dimensional) Sierpinski curves, vertex-labelling algorithms will prove to be a comfortable choice, though, as a d-dimensional simplex only has $d + 1$ vertices – see Chap. 12.

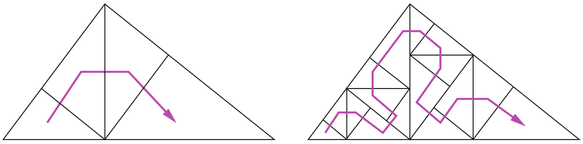
What's next?

- ➔ The next chapter will provide some further examples of space-filling curves – to get some more practice with the different mathematical tools to describe space-filling curves. You may safely skip it, or leave it for later.
- ↘ The default is then to move on with Chap. 8, which will discuss 3D space-filling curves.
- ↱ Are you wondering how a Sierpinski curve might look in 3D? This is a more complicated question: a first option is presented in Sect. 8.3, however, this curve is not much used in practice. We will discuss the whole complex in Chap. 12.

Exercises

6.1. As indicated in Figs. 6.2 and 6.3, the Sierpinski curve can also be constructed via bisection of triangles. Derive grammar representations and arithmetisations for the Sierpinski curve following the bisection-based construction.

6.2. The plot below illustrates Polya's extension of the Sierpinski curve, which is constructed on right triangles. Discuss whether and how Polya's curve could be described via an arithmetisation or via grammars.



6.3. In Sect. 3.4, we introduced turtle-based traversals for the Hilbert curve. Derive a corresponding turtle-based traversal (and respective grammar) for the Sierpinski curve.

6.4. Consider the bisection-based construction from Exercise 6.1 for generalised Sierpinski curves. Show that we now obtain eight congruency classes for the triangles!

6.5. Change Algorithm 6.2 into an algorithm that computes the Hilbert mapping, i.e., the image point for a given parameter.

6.6. Generate a vertex-labeling algorithms for different variants of the Peano curve.

Chapter 7
Further Space-Filling Curves

7.1 Characterisation of Space-Filling Curves

During the construction of the Hilbert curve, the various Peano curves, and also of the Sierpinski curve, we have consistently followed two substantial principles of construction:

1. Starting from regularly refined sequences of nested intervals and nested subsquares or subtriangles, we mapped intervals to corresponding square or triangle subdomains.
2. Squares or triangles that belong to adjacent intervals were again adjacent, i.e. they had at least a common edge.

For further space-filling curves generated via this principle, including 3D and higher-dimensional curves (see Chap. 8), we will therefore formalise these construction principles via the following definitions.

Definition 7.1 (*Recursive* Space-Filling Curve). A space-filling curve $f\colon\mathcal{I} \to \mathcal{Q} \subset \mathbb{R}^d$ is called *recursive*, if there is a multi-level partitioning of both $\mathcal{I}$ and $\mathcal{Q}$ into subintervals $\mathcal{I}_{l,m}$ and subdomains $\mathcal{Q}_{l,m}$ (with $\mathcal{I} = \mathcal{I}_{0,0}$ and $\mathcal{Q} = \mathcal{Q}_{0,0}$), such that

- On each level $l > 0$, the intervals $\mathcal{I}_{l,m}$ and the subdomains $\mathcal{Q}_{l,m}$ result from a partitioning of the parent intervals $\mathcal{I}_{l-1,\hat{m}}$ and parent domains $\mathcal{Q}_{l-1,\hat{m}}$ into $M(l)$ parts of equal size (i.e. equal area or volume of the $\mathcal{Q}_{l,m}$);
- The subdomains $\mathcal{Q}_{l,m}$ are congruent to their parent domains $\mathcal{Q}_{l-1,\hat{m}}$ up to a suitable scaling;
- The subintervals $\mathcal{I}_{l,m}$ are mapped to the corresponding subdomains $\mathcal{Q}_{l,m}$, i.e.: $f_*(\mathcal{I}_{l,m}) = \mathcal{Q}_{l,m}$ for all l and m.

With the term *partitioning*. we mean that neither the intervals $\mathcal{I}_{l,m}$ nor the subdomains $\mathcal{Q}_{l,m}$ of a common level l will overlap, i.e., will at most share common boundaries, and that the union of all child intervals or subdomains will be equal

M. Bader, *Space-Filling Curves*, Texts in Computational Science and Engineering 9,
DOI 10.1007/978-3-642-31046-1_7, © Springer-Verlag Berlin Heidelberg 2013

to their parent. This requires that the subdomains $\mathcal{Q}_{l,m}$ are so-called *space-fillers*, i.e., geometrical objects that can be seamlessly arranged to fill the entire plane (in 2D) or space.

Definition 7.2 (*Connected* Space-Filling Curves). A d-dimensional recursive space-filling curve is called *connected*, if any two subsequent subdomains $\mathcal{Q}_{l,m}$ and $\mathcal{Q}_{l,n}$, i.e. subdomains that are images of adjacent intervals $\mathcal{I}_{l,m}$ and $\mathcal{I}_{l,n}$, share a common, $(d-1)$-dimensional boundary face.

In particular, we call a space-filling curve *edge-connected*, if subsequent subdomains at least share a common edge, or (esp. in 3D) *face-connected*, if subsequent subdomains share a common face. We will call a space-filling curve *node-connected*, if subsequent subdomains at least share one common point.

The Hilbert curve, as well as all Peano curves discussed so far (including the Peano-Meander curves) are therefore connected, recursive space-filling curves. We will also demand these properties for the higher-dimensional variants. In particular, 3D Hilbert or Peano curves should be face-connected (see Chap. 8). Connected, recursive space-filling curves always have the following properties:

- The respective mapping functions are *uniformly continuous*, as connectedness and recursivity is sufficient for the respective proof (compare Sect. 2.3.5). In Sect. 11.1, we will show that those properties are even sufficient to prove the stricter Hölder continuity (with exponent $1/d$).
- The respective curves are *locality preserving* – again this is a result of connectedness and recursivity; we will see that the respective locality properties can be quantified by the Hölder continuity.
- Their iterations can be *described via grammars* – hence, we can derive a respective traversal algorithm following the standard procedure.
- The mapping may be *computed via an arithmetisation* – which provides us with algorithms to computer curve points from parameters and vice versa.

To define a space-filling curve via an arithmetisation, we do not necessarily need the connectedness – as the example of Morton order will show in Sect. 7.2. However, if subsequent subdomains are not even connected via a common point, we will normally not obtain a continuous mapping.

Definition 7.3 (*Simple* Space-Filling Curves). Finally, we will call a space-filling curve *simple*, if an arithmetisation, as presented in Chap. 4, uses the same set of operators for each recursive level of the construction.

Hence, the Hilbert, Sierpinski, and Peano curve are all simple space-filling curves. The Hilbert-Moore curve is an example of a non-simple curve. Any recursive and simple space-filling curve necessarily has to be *self-similar*. However, also non-simple, recursive space-filling curves will be self-similar, if their grammar representation requires only a limited set of basic patterns.

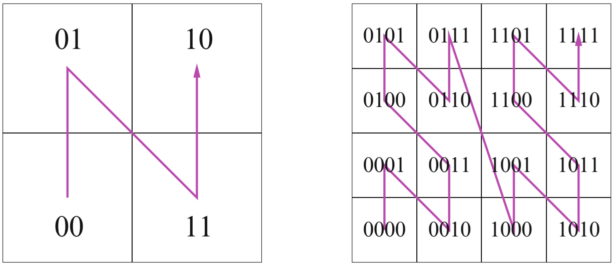


Fig. 7.1 Substructuring and sequentialisation of subsquares used for Morton order. Each subsquare contains the first digits of the binary representation of the parameters that are mapped to this square

7.2 Lebesgue Curve and Morton Order

Already in Sect. 2.1, we discussed a simple mapping that maps the unit interval to the unit square: the mapping exploits the binary representation of the parameter t,

$$m(t) = m(0_2.b_1b_2b_3b_4b_5b_6\ldots) = \begin{pmatrix} 0_2.b_1b_3b_5\ldots \\ 0_2.b_2b_4b_6\ldots \end{pmatrix}, \qquad (7.1)$$

and is also known as *Morton order*, especially in computer science.

All image points that share the same start sequence of binary digits, for example

$$m(t) = \begin{pmatrix} 0_2.010\ldots \\ 0_2.101\ldots \end{pmatrix},$$

lie in a common subsquare of side length 2^{-3} (in general 2^{-n}, if they share the first n bits). The respective subsquare is generated by performing the recursive splitting of a square domain into four subsquares three times (similar to the recursive construction of the Hilbert curve). The parameters that are mapped to this square are of the form

$$t = 0_2.011001\ldots$$

All these parameters are within an interval of length 4^{-3}, and again such an interval is generated via generation of nested intervals similar to the Hilbert construction.

Thus, the Morton order can be defined via nested intervals and nested squares, as well. However, the respective subsquares are sequentialised in a different order by the Morton scheme. Figure 7.1 illustrates the generated sequentialisation. We observe that the sequentialisation is of a much simpler type than the Hilbert curve. In particular, the sequentialisation pattern is identical in each subsquare – starting in the lower-left corner, and proceeding to the top-left, lower-right, and top-right corner.

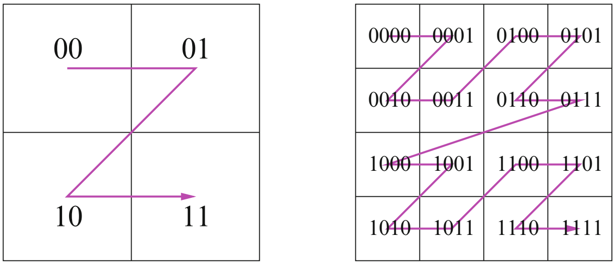


Fig. 7.2 Sequential order of subsquares generated by the Z-curve

In contrast to the Hilbert mapping, the mapping $m(t)$ generated by the Morton order is obviously not continuous. For example, a jump occurs at $t = \frac{1}{2}$. There, the left-sided limit is

$$\lim_{n\to\infty} m(0_2.0\underbrace{111\ldots111}_{2n-1 \text{ times}}) = \lim_{n\to\infty} \begin{pmatrix} 0_2.011\ldots1 \\ 0_2.\underbrace{111\ldots1}_{n \text{ times}} \end{pmatrix} = \begin{pmatrix} \frac{1}{2} \\ 1 \end{pmatrix},$$

while the right-sided limit is given as

$$m(0_2.1000\ldots) = \lim_{n\to\infty} \begin{pmatrix} 0_2.1000\ldots \\ 0_2.000\ldots \end{pmatrix} = \begin{pmatrix} \frac{1}{2} \\ 0 \end{pmatrix}.$$

Z-Curve

If we exchange the coordinates in the definition of the Morton mapping $m(t)$, we obtain the coordinate function

$$z(t) = z(0_2.b_1b_2b_3b_4b_5b_6\ldots) := \begin{pmatrix} 0_2.b_2b_4b_6\ldots \\ 0_2.b_1b_3b_5\ldots \end{pmatrix}. \tag{7.2}$$

The sequentialisation of subsquares generated by this mapping is then congruent to the Morton order given in Fig. 7.1, but reflected at the main diagonal. Because of the generated numbering patterns, which are illustrated in Fig. 7.2, the respective function is often called the *Z-function* or *Z-order*. Similarly, we could also call the Morton order N-order, and could define similar other orders, as well. The naming, of course, might change if the origin of the coordinate system is placed in a different corner than the lower-left corner, as done throughout this book. We will therefore use the term Morton order as a generic term that can reflect Z-order, N-order, or (esp. in higher dimensions) any other scheme that results from bit interleaving schemes.

Fig. 7.3 Recursive construction of the Cantor Set by removing the *middle* thirds

The Lebesgue Curve and the Cantor Set

Morton order, Z-curve, and relatives cannot really be called space-filling curves, as they lack the continuity required for a curve. However, we can turn them into a real space-filling curve using a certain trick that will restore continuity – the trick is to change the index set for the parameters. If at every point, where the Z-curve or its relatives have a jump, the index set would have a jump, too, the continuity would no longer be violated at this position. For a 1D index interval, such a "jump" can only be a gap in the interval. Hence, we require an index set with enough gaps (actually an infinite number of gaps), which also has to follow a similar recursive construction principle as the corresponding curve.

A well-known example of such an index set is the so-called *Cantor Set*. It is generated by recursively splitting intervals into three parts and removing the (open) subinterval in the centre (starting with the unit interval). All points that survive this recursive cutting procedure are defined to form the Cantor Set. Figure 7.3 illustrates this construction principle. Memorising our considerations of the previous chapters, it is not surprising that the Cantor Set can also be characterised via ternaries:

Definition 7.4 (Cantor Set). The set of all numbers $t \in [0, 1]$ that can be represented by ternaries of the form

$$t = 0_3, t_1 t_2 t_3 \ldots, \qquad t_i \in \{0, 2\},$$

is called the *Cantor Set* $\mathcal{C}$.

Hence, the removal of middle thirds is achieved by disallowing the digit 1 in the ternary representation. All numbers that contain a 1-digit in their ternary representation are excluded from the set – with the exception of finite ternaries that end with a one, as these can be represented via an infinite ternary:

$$0_3.t_1 \ldots t_n 1 = 0_3.t_1 \ldots t_2 0222 \ldots$$

On this basis, we can define a space-filling curve analogous to the Z-curve:

Definition 7.5 (Lebesgue Curve). The image of the mapping b from the Cantor Set $\mathcal{C}$ into the unit square $[0, 1]^2$, with

$$b : \mathcal{C} \to [0, 1]^2, \quad b(t) = l(0_3, t_1 t_2 t_3 t_4 \ldots) = \begin{pmatrix} 0_2, t_1/2\; t_3/2 \ldots \\ 0_2.t_2/2\; t_4/2 \ldots \end{pmatrix},$$

is called *Lebesgue Curve*.

The Lebesgue curve is surjective, as each point of the unit square,

$$p(t) = \begin{pmatrix} 0_2.b_1b_2b_3\dots \\ 0_2.d_1d_2d_3\dots \end{pmatrix} \in [0,1]^2,$$

is an image of the parameter

$$t = 0_3.(2b_1)\,(2d_1)\,(2b_2)\,(2d_2)\,(2b_3)\,(2d_3)\dots,$$

which is obviously a member of the Cantor Set $\mathcal{C}$.

Continuity of the Lebesgue Curve

For the Lebesgue curve, we can show that it is continuous in every point $t_0 \in \mathcal{C}$. Let $t \in \mathcal{C}$ be a parameter with $|t - t_0| < 3^{-2n}$, where 3^{-2n} is exactly the width of the delete middle third in the $2n$-th iteration during construction of the Cantor Set. If t_0 and t are closer than 3^{-2n}, then there cannot be a respective gap between them. As a consequence, their first $2n$ ternary digits need to be identical:

$$\begin{aligned} t_0 &= 0_3.t_1t_2\dots t_{2n}\dots \\ t &= 0_3.t_1t_2\dots t_{2n}\dots \end{aligned}$$

For the corresponding images on the Lebesgue curve, we therefore obtain that

$$\begin{aligned} b(t_0) &= \begin{pmatrix} 0_2.t_1t_3\dots t_{2n-1}\dots \\ 0_2.t_2t_4\dots t_{2n}\dots \end{pmatrix} \\ b(t) &= \begin{pmatrix} 0_2.t_1t_3\dots t_{2n-1}\dots \\ 0_2.t_2t_4\dots t_{2n}\dots \end{pmatrix}. \end{aligned}$$

The two points share the first n digits and therefore lie in a common subsquare with side length 2^{-n}. Their maximum possible distance is therefore given by the length of the diagonal of this subsquare:

$$\|b(t_0) - b(t)\| \le 2^{-n}\sqrt{2}.$$

Hence, for a given $\varepsilon > 0$, we can choose an n such that $2^{-n}\sqrt{2} < \varepsilon$. With $\delta = 3^{-2n}$, we can infer that for all t with $|t - t_0| < \delta$ the inequality $\|b(t_0) - b(t)\| < \varepsilon$ holds due to our previous calculation. This exactly implies the continuity of b at t_0. As t_0 was chosen arbitrarily, b is continuous on the entire index set $\mathcal{C}$.

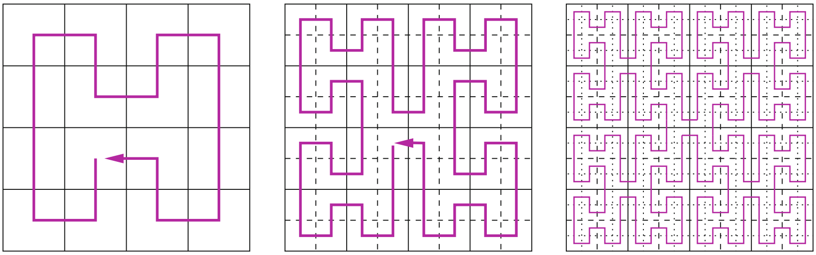


Fig. 7.4 Sequential orders given by the H-index (on 4 × 4, 8 × 8, and 16 × 16 elements)

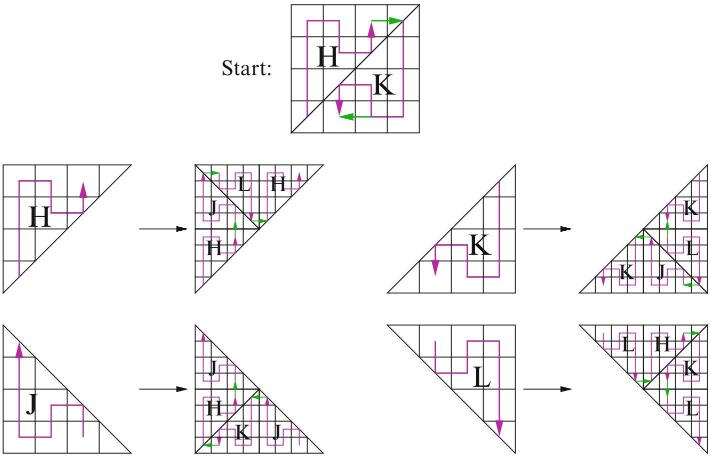


Fig. 7.5 Derivation of the grammar for the H-index

7.3 The *H*-Index

The H-index (or H-order) was introduced as a sequentialisation for the elements of a $2^n \times 2^n$ array. Respective element orders are illustrated in Fig. 7.4. Even though it is intended for quadratic domains, its construction is based on triangles, as illustrated in Fig. 7.5. From the iteration rules depicted in Fig. 7.5, we can easily derive the following productions for a grammar representation of the H-index:

$$\begin{array}{rcl|l}
H & \longleftarrow & H \uparrow J \rightarrow L \rightarrow H & \uparrow\uparrow\uparrow\rightarrow\downarrow\rightarrow\uparrow \\
J & \longleftarrow & J \rightarrow K \uparrow H \leftarrow J & \uparrow\leftarrow\downarrow\leftarrow\uparrow\uparrow\uparrow \\
K & \longleftarrow & K \leftarrow L \downarrow J \rightarrow K & \downarrow\downarrow\downarrow\leftarrow\uparrow\leftarrow\downarrow \\
L & \longleftarrow & L \downarrow H \leftarrow K \uparrow L & \downarrow\rightarrow\uparrow\rightarrow\downarrow\downarrow\downarrow
\end{array}$$

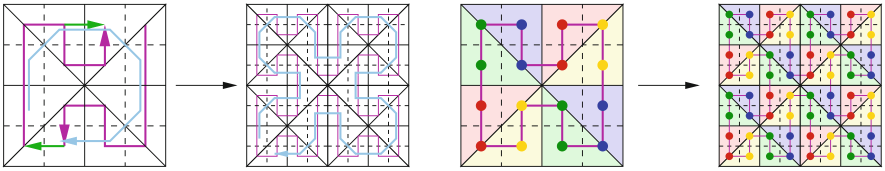


Fig. 7.6 Comparison of the H-order with the Sierpinski curve

A close examination of the terminal productions reveals that we can replace these by simpler productions that correspond to only one step:

$$\begin{array}{rcl@{\quad}c@{\quad}c}
H & \longleftarrow & H \uparrow J \rightarrow L \rightarrow H & | & \uparrow \\
J & \longleftarrow & J \rightarrow K \uparrow H \leftarrow J & | & \uparrow \\
K & \longleftarrow & K \leftarrow L \downarrow J \rightarrow K & | & \downarrow \\
L & \longleftarrow & L \downarrow H \leftarrow K \uparrow L & | & \downarrow
\end{array}$$

One expansion step on the non-terminals immediately brings back the original productions.

From the H-Order to the Sierpinski Curve

If we remember the grammar-based description of the Sierpinski curve,

$$\begin{array}{rcl}
S & \longleftarrow & S \nearrow R \rightarrow P \searrow S \\
R & \longleftarrow & R \nwarrow Z \uparrow S \nearrow R \\
Z & \longleftarrow & Z \swarrow P \leftarrow R \nwarrow Z \\
P & \longleftarrow & P \searrow S \downarrow Z \swarrow P
\end{array}$$

we recognise a strong similarity to the grammar for the H-index. The recursion scheme for the non-terminals is equivalent, if we map the symbols $S \rightarrow H$, $R \rightarrow J$, $Z \rightarrow K$, and $P \rightarrow L$. This structural equivalence is due to the triangle-based construction: in both constructions, the triangles are subdivided according to the same scheme and the geometrical orientation of the triangles determines the pattern of the curve. In addition, the child triangles are traversed in the same order – which is not too surprising, as there is, in fact, no other choice, if we require adjacent triangles to share a common edge.

Figure 7.6 shows a comparison of the H-index with the iterations (left image) and with the approximating polygons (right image) of the Sierpinski curve. We can obviously obtain the H-index traversal by replacing all diagonal steps of the Sierpinski iteration by two steps that are in horizontal and vertical direction. And we

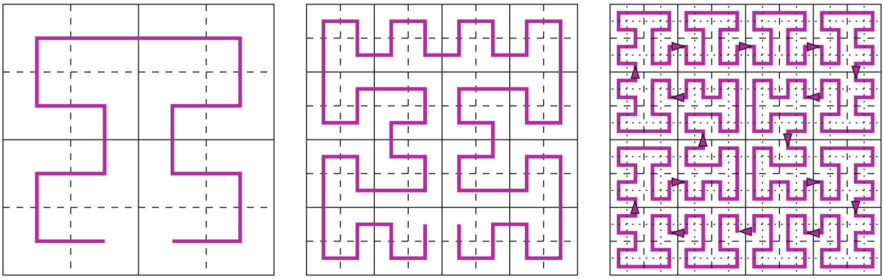


Fig. 7.7 The second, third, and fourth iteration of the $\beta\Omega$-curve. Note the *arrows* in the fourth iteration, which mark the transfer points between the subsquares

also see that all nodes of the H-index iteration are also nodes of the approximation polygon of the Sierpinski curve. Moreover these H-index nodes are visited in Sierpinski order. We can therefore interpret the H-index iterations as polygons that approximate the Sierpinski curve, which implies that the H-index will not lead to a new space-filling curve. Its infinite iteration will lead to the Sierpinski curve, again.

7.4 The $\beta\Omega$-Curve

The $\beta\Omega$-curve is a variation of the Hilbert curve, as it is based on the same elementary patterns. However, while for the Hilbert curve the refinement process in each step is identical up to rotations and reflections, the $\beta\Omega$-curve uses a more complicated refinement algorithm. Similar to Moore's variant of the Hilbert curve, the $\beta\Omega$-curve is constructed as a closed curve. The second to fourth iteration of the $\beta\Omega$-curve is illustrated in Fig. 7.7.

As can be observed in the fourth iteration, the entry and exit points of the curve into and out of the individual subsquares are no longer situated in the corners, but on the edges. As we will see, these points divide the edges according to the ratio 1:2 or 2:1. Hence, their distance to the nearest corner is one third of the side length. We also observe that entry and exit points can be placed both on adjacent edges and opposing edges, which leads to two different patterns – referred to a β and Ω, respectively.

The exact construction of the $\beta\Omega$-curve is difficult to describe in words, so we will use a grammar-based description, right away. Figure 7.8 illustrates the refinement scheme for the $\beta\Omega$-curve, which uses six different patterns:

- Ω and Ψ represent the patterns where entry and exit points are across. For Ω, the entry and exit point is closer to the right-hand corners.
- B and D are two patterns that lead to a left turn, i.e. the exit edge is adjacent top the right-hand corner of the entry edge. Correspondingly, C and E lead to right turns.
- While in B and E the entry point is closer to the right-hand corner of the entry edge, the entry point of C and D is closer to the left-hand corner.

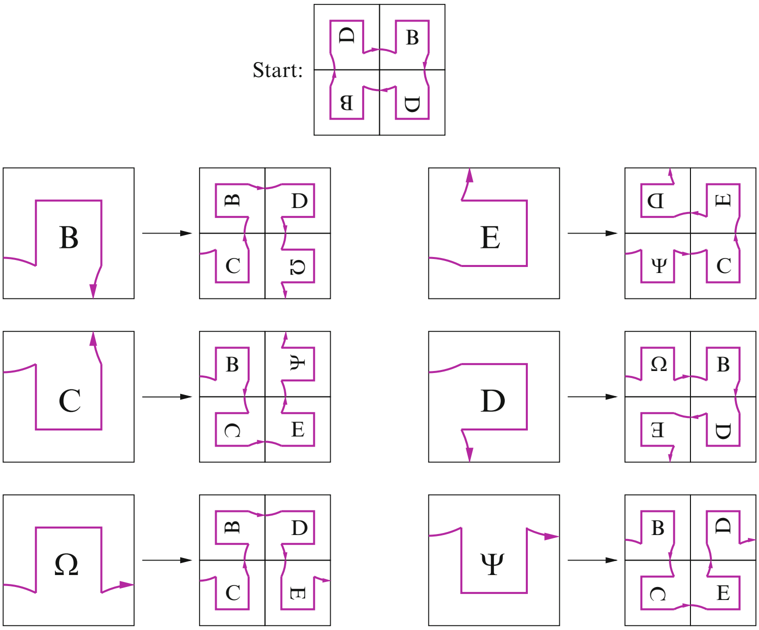


Fig. 7.8 Derivation of the grammar for the $\beta\Omega$-curve

As can be seen from Fig. 7.8, the six patterns form a closed system to refine the patterns to arbitrary resolution. Different orientations of the patterns are required, which is indicated by rotated characters.

In Fig. 7.8, we took up the turtle-grammar concept of Sect. 3.4, using only relative movements (“move forward”, “turn left”, “turn right”), but no absolute directions, to describe the iterations. Hence, our grammar for the $\beta\Omega$-curve will consist of the non-terminals $\{\beta, \Omega, \Psi, B, C, D, E\}$ (β is only used as the starting symbol) and of the terminal characters $\{\uparrow, \Lsh, \Rsh\}$. In the same way as for the Hilbert grammar (see Sect. 3.4), we can derive the following productions from Fig. 7.8:

$$\begin{array}{rcll}
\beta & \longleftarrow & D \uparrow B \uparrow D \uparrow B \uparrow & \\
B & \longleftarrow & C \uparrow B \uparrow D \uparrow \Omega & \mid \ \Rsh \\
C & \longleftarrow & B \uparrow C \uparrow E \uparrow \Psi & \mid \ \Lsh \\
D & \longleftarrow & \Omega \uparrow B \uparrow D \uparrow E & \mid \ \Rsh \\
E & \longleftarrow & \Psi \uparrow C \uparrow E \uparrow D & \mid \ \Lsh \\
\Omega & \longleftarrow & C \uparrow B \uparrow D \uparrow E & \mid \ \varepsilon \\
\Psi & \longleftarrow & B \uparrow C \uparrow E \uparrow D & \mid \ \Rsh
\end{array}$$

Note that the last $\uparrow$ in the production for β closes the curve. It is now straightforward to produce a traversal algorithm from this grammar. A plot of the sixth iteration of the $\beta\Omega$-curve is given in Fig. 7.9.

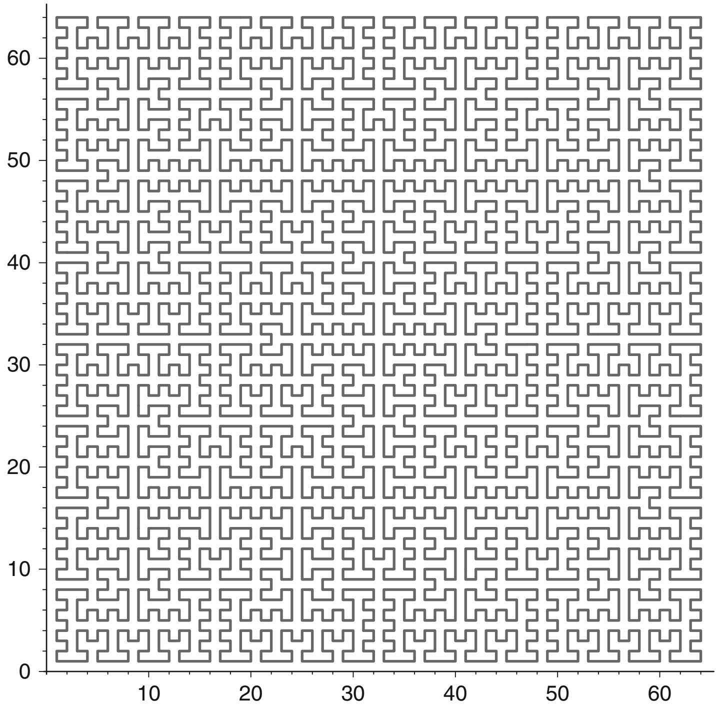


Fig. 7.9 The sixth iteration of the $\beta\Omega$-curve

Arithmetisation of the $\beta\Omega$-Curve

For the Hilbert curve, we determined an arithmetic representation of the Hilbert mapping h via the recursion

$$h(0_4.q_1q_2q_3\ldots) = H_{q_1} \circ h(0_4.q_2q_3\ldots).$$

For the $\beta\Omega$-curve, the situation is again more complicated. If we examine the replacement schemes in Fig. 7.8, we quickly notice that the patterns represented by Ω and Ψ require different operators than B and C. And while B and C are just reflections of each other, D and E again require a different set of operators. They would only be congruent to B or C, if we were able to reverse the orientation.

An arithmetisation is possible, nevertheless, if we define four different mappings (each of which has a separate set of operators):

- γ will describe the mapping for the patterns B and C;
- δ will describe the mapping for D and E;
- ω will describe the mapping for Ω and Ψ (with operators $G_0, \ldots, G_3$).
- β, finally, will be the $\beta\Omega$-mapping – which has to take care of the starting configuration.

Table 7.1 Operators to compute the $\beta\Omega$-mapping

i	0	1	2	3
$f_i^{(\beta)}$	δ	γ	δ	γ
$Q_i^{(\beta)}$	$\left(\begin{array}{cc\|c} 0 & -\frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & 0 & 0 \end{array}\right)$	$\left(\begin{array}{cc\|c} 0 & -\frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & 0 & \frac{1}{2} \end{array}\right)$	$\left(\begin{array}{cc\|c} 0 & \frac{1}{2} & \frac{1}{2} \\ -\frac{1}{2} & 0 & 1 \end{array}\right)$	$\left(\begin{array}{cc\|c} 0 & \frac{1}{2} & \frac{1}{2} \\ -\frac{1}{2} & 0 & \frac{1}{2} \end{array}\right)$
$f_i^{(\gamma)}$	γ	γ	δ	ω
$Q_i^{(\gamma)}$	$\left(\begin{array}{cc\|c} \frac{1}{2} & 0 & 0 \\ 0 & -\frac{1}{2} & \frac{1}{2} \end{array}\right)$	$\left(\begin{array}{cc\|c} 0 & -\frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & 0 & \frac{1}{2} \end{array}\right)$	$\left(\begin{array}{cc\|c} 0 & \frac{1}{2} & \frac{1}{2} \\ -\frac{1}{2} & 0 & 1 \end{array}\right)$	$\left(\begin{array}{cc\|c} 0 & \frac{1}{2} & \frac{1}{2} \\ -\frac{1}{2} & 0 & \frac{1}{2} \end{array}\right)$
$f_i^{(\delta)}$	ω	γ	δ	δ
$Q_i^{(\delta)}$	$\left(\begin{array}{cc\|c} 0 & -\frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & 0 & 0 \end{array}\right)$	$\left(\begin{array}{cc\|c} 0 & -\frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & 0 & \frac{1}{2} \end{array}\right)$	$\left(\begin{array}{cc\|c} 0 & \frac{1}{2} & \frac{1}{2} \\ -\frac{1}{2} & 0 & 1 \end{array}\right)$	$\left(\begin{array}{cc\|c} \frac{1}{2} & 0 & \frac{1}{2} \\ 0 & -\frac{1}{2} & \frac{1}{2} \end{array}\right)$
$f_i^{(\omega)}$	γ	γ	δ	δ
$Q_i^{(\omega)}$	$\left(\begin{array}{cc\|c} \frac{1}{2} & 0 & 0 \\ 0 & -\frac{1}{2} & \frac{1}{2} \end{array}\right)$	$\left(\begin{array}{cc\|c} 0 & -\frac{1}{2} & \frac{1}{2} \\ \frac{1}{2} & 0 & \frac{1}{2} \end{array}\right)$	$\left(\begin{array}{cc\|c} 0 & \frac{1}{2} & \frac{1}{2} \\ -\frac{1}{2} & 0 & 1 \end{array}\right)$	$\left(\begin{array}{cc\|c} \frac{1}{2} & 0 & \frac{1}{2} \\ 0 & -\frac{1}{2} & \frac{1}{2} \end{array}\right)$

For each of these mappings, we require a set of operators, which we will denote as $Q_i^{(\gamma)}$, $Q_i^{(\delta)}$, $Q_i^{(\omega)}$, and $Q_i^{(\beta)}$. The indices $i = 0, \dots 3$, as usual, will be given by the respective quaternary digits. In addition, the recursive calls of each mapping will also depend on the digits – i.e., a different mapping will be called in each step, such as:

$$\beta(0_4.q_1 q_2 q_3 \dots) = Q_{q_1}^{(\beta)} \circ f_{q_1}^{(\beta)}(0_4.q_2 q_3 \dots),$$

where $f_i^{(\beta)} \in \{\gamma, \delta, \omega\}$ (β itself is not called recursively, as it reflects the starting pattern).

Table 7.1 lists all operators and also specifies the recursive calls depending on the quaternary digits. Function `betaOmega`, finally gives an algorithm to compute the $\beta\Omega$-mapping. It takes *fun* as a parameter to specify the four different mappings γ, δ, ω, and β. The operators $Q_q^{(fun)}$ and the recursive scheme of mappings $f_q^{(fun)}$ are taken from Table 7.1.

7.5 The Gosper Flowsnake

So far, all our space-filling curves were based on tesselations of the plane into triangles or rectangles. In this section, we will discuss a space-filling curve, the so-called Gosper curve (or Gosper flowsnake), that is based on a tesselation into hexagons. The construction is illustrated in Fig. 7.10, which plots the first two

Function betaOmega – compute the $\beta\Omega$-curve (fixed digits)

```
Function betaOmega (fun, t, depth)
    Parameter: fun: type of $\beta\Omega$ mapping, $fun \in \{\gamma, \delta, \omega, \beta\}$ ($fun = \beta$ on entry)
               t,depth: index parameter and depth of recursion (as usual)
begin
    if depth = 0 then
        return (0,0)
    else
        // compute next quaternary digit in q
        q := floor(4*t);
        r := 4*t - q;
        // recursive call
        (x,y) := betaOmega($f_q^{(fun)}$, r, depth-1);
        // apply operator $Q_q$ for correct mapping fun:
        return $Q_q^{(fun)}(x, y)$;
    end
end
```

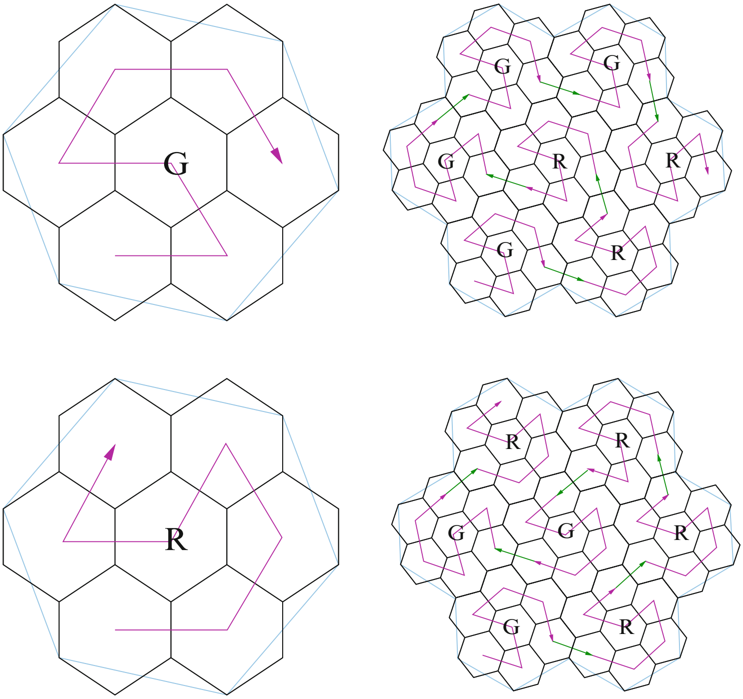


Fig. 7.10 The first two iterations of the Gosper curve, G and R representing the two basic generating patterns

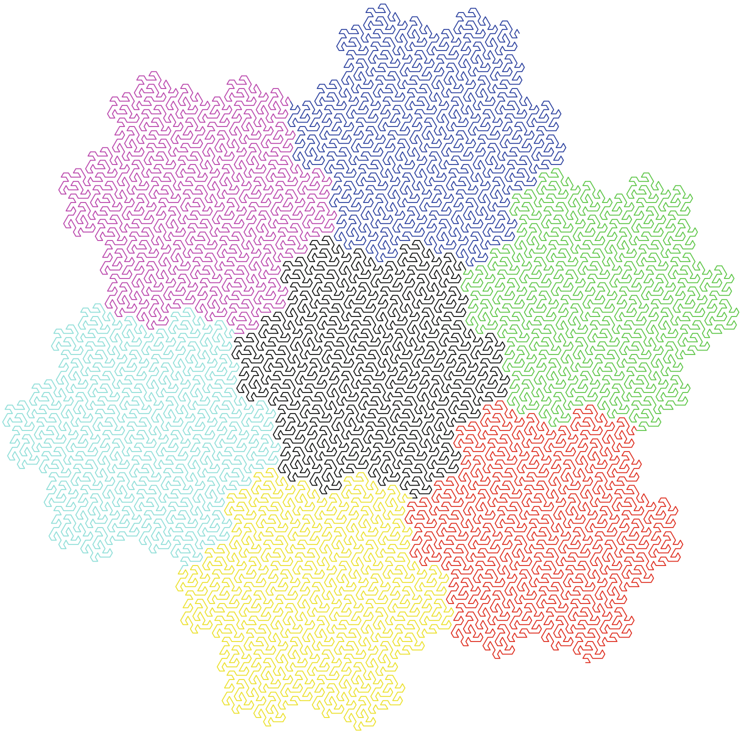


Fig. 7.11 The fifth iteration of the Gosper curve. The seven connected fourth iterations, each filling a smaller approximated Gosper island, are plotted in different colours

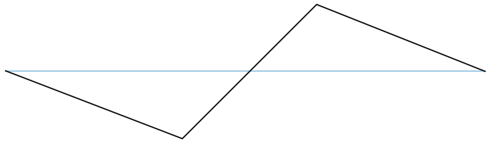
iterations of the Gosper curve. From this illustration, we can extract the following grammar description:

$$\begin{aligned} G &\longleftarrow Gl \uparrow Rl \uparrow R \uparrow rG \uparrow rGl \uparrow G \uparrow rR \\ &\quad\mid \ \uparrow ll \uparrow l \uparrow rr \uparrow r \uparrow r \uparrow \\ R &\longleftarrow Gl \uparrow R \uparrow rRl \uparrow Rl \uparrow G \uparrow rG \uparrow rR \\ &\quad\mid \ \uparrow l \uparrow l \uparrow ll \uparrow r \uparrow rr \uparrow \end{aligned} \tag{7.3}$$

with non-terminals G and R, and $\{\uparrow, r, l\}$ as terminal characters. G and R represent the two basic patterns, as indicated in Fig. 7.10, but do not reflect the rotation of the patterns. Consequently, we follow the turtle-graphics concept for the terminal characters, such that $\uparrow$ models a forward step and $\{r, l\}$ prescribe 60°-rotations to the left or right, respectively. Figure 7.11 shows the fifth iteration of the Gosper curve, as it results from a straightforward implementation of the grammar in Eq. (7.3).

Obviously, the Gosper curve does not fill a hexagon. As a hexagon cannot be split into smaller, congruent hexagons, the subdomains have to be slightly modified in each iteration, such that they can be split into seven congruent subdomains. In

Fig. 7.10, this change of domains is illustrated via a light blue line that represents the boundary of the previous hexagonal cells. The subsequent modification of the boundary lines leads to a domain that has a fractal curve as boundary, which is generated by the following generator:



The resulting fractal-bounded domain (with a hexagon as initiator) is called the Gosper island – in Fig. 7.11, we can see how the iterations of the Gosper curve successively fill the Gosper island, and how the Gosper island is substructured into seven smaller islands. Thus, the Gosper island is a fractal space-filler.

References and Further Readings

The classification of space-filling curves in the given form was already introduced by Wunderlich [277], who pointed out that to construct space-filling curves, these should be *recursive* and *connected* according to the definition given in Sect. 7.1 (where he also assumed the curves to be *simple* and use a uniform set of transformations). In addition, he required that exit and entry point of the space-filling curve in neighbouring subdomains are identical.

Lebesgue introduced his curve in a textbook on integration and analysis of functions [157], using a notation very similar to Definition 7.5. He already pointed out that the curve is trivially extended to higher dimensions. The term "Morton order" refers to a technical report by Morton [194], who described the application of bit interleaving for codes in the context of geographical data bases – which is likely to be the first application of Morton order in computer science. The Lebesgue curve and Morton order found their way into Scientific Computing with the use of quadtree and octree data structures – see the respective references of Chap. 9.

The H-index was introduced by Niedermeier et al. [196], who proved, in the same paper, that the H-index has optimal locality properties among all cyclic indexings (which derive from closed space-filling curves). Similarly, the $\beta\Omega$-curve, found by Wierum [268], was developed as a curve with better locality properties than the Hilbert curve. It is slightly worse than the H-index, though, but – in contrast to H-index – is a recursive space-filling curve. See also Chap. 11 on locality properties of space-filling curves.

The "flowsnake" discussed in Sect. 7.5 was constructed by William Gosper in 1975, and first presented a year later by Martin Gardner [95] in his series on "Mathematical Games". This article also discussed the generation of the Gosper island (see Exercise 7.5). Both, island and curve, were also discussed by Mandelbrot [174].

Fukuda et al. [93] and Akiyama et al. [6] introduced a multitude of variants of the Gosper curve with varying degree (i.e., number of recursive subdomains) using a construction based on equilateral triangles. The Gosper curve rarely shows up in applications, though it would be an obvious choice for meshes constructed from hexagons.

What's next?

- The next chapter will take us to the three-dimensional case, and thus to "true" space(!)-filling curves.

Exercises

7.1. Derive a standard arithmetisation of Morton order, and examine how it corresponds to the bit-oriented definition of the mapping.

7.2. Due to the structural similarity of the H-index and the Sierpinski curve, we can use the arithmetisation of the Sierpinski curve to compute the H-index of a given grid cell (and vice versa). Describe how the respective algorithm to compute the Sierpinski index (or Sierpinski mapping) has to be modified.

7.3. Draw the approximating polygons for the Gosper curve and derive a grammar to describe them.

7.4. Derive an arithmetisation of the Gosper curve.

7.5. Try to compute the area of the Gosper island by computing the areas enclosed by the successive boundary polygons. For the areas of the seven child-islands, we expect 1/7-th of the area of the parent island. On the other hand, one can construct the boundary of a Gosper island by connecting six half-boundaries of the child-islands, which suggests that the boundary of the child islands is one third of that of the parent island. Is there a conflict?

Chapter 8
Space-Filling Curves in 3D

8.1 3D Hilbert Curves

To construct a three-dimensional Hilbert curve, we want to retain the characteristic properties of the 2D Hilbert curve in 3D. Hence, the 3D Hilbert curve should

- Fill the unit cube instead of the unit square;
- Be recursive, and should be based on recursive substructuring of cubes into subcubes with halved side length – hence, a substructuring into eight subcubes is wanted;
- Be face-connected.

These requirements leave us with three possible basic patterns, which are illustrated in Fig. 8.1.

While the first two patterns, similar to the 2D construction, start and end at a common edge of the cube, the “spiral form” depicted in the right-most plot has its start and end points diagonally across, instead. Such a scenario did not occur in 2D. As this diagonal curve will lead to a different kind of approximating polygons, we will not consider it for the construction of a “proper” Hilbert curve – in particular, it can be shown that such a construction cannot lead to a *simple* space-filling curve. We will also disregard the pattern in the centre, and concentrate on the left-most pattern, which is most similar to the 2D pattern.

8.1.1 *Possibilities to Construct a 3D Hilbert Curve*

Even if we restrict ourselves to the left-most pattern in Fig. 8.1, the construction of a 3D Hilbert curve is not straightforward as in the 2D case. We will not obtain a unique “Hilbert Curve”, but a whole collection of curves that can all be called a 3D Hilbert curve. In the following, we will discuss the possible variants.

M. Bader, *Space-Filling Curves*, Texts in Computational Science and Engineering 9,
DOI 10.1007/978-3-642-31046-1_8, © Springer-Verlag Berlin Heidelberg 2013

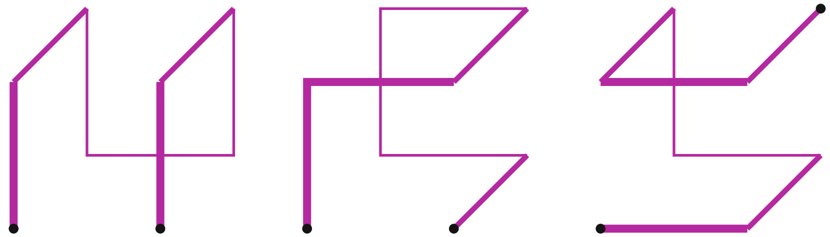


Fig. 8.1 Possible basic patterns to construct a 3D Hilbert curve

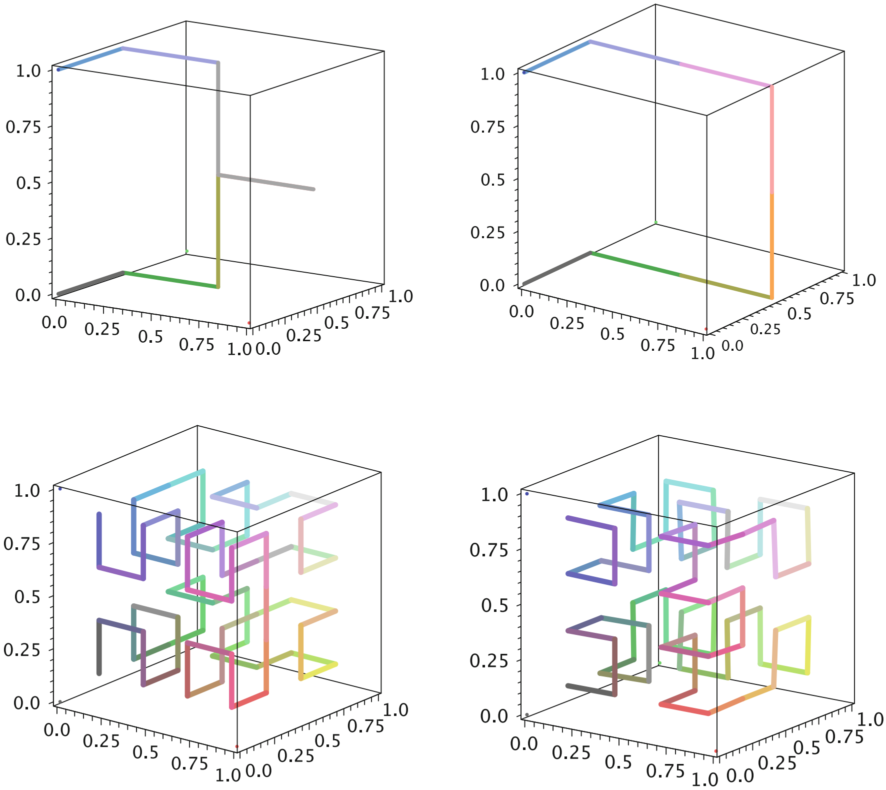


Fig. 8.2 Two variants from which to choose the approximating polygon of a 3D Hilbert curve (*top-left* and *top-right* plot). The plots below show the corresponding second iteration of the respective Hilbert curves

Varying the Approximating Polygon

Figure 8.2 illustrates two different constructions of 3D Hilbert curves. In both constructions, the eight subcubes of the first iteration are sequentialised in the same

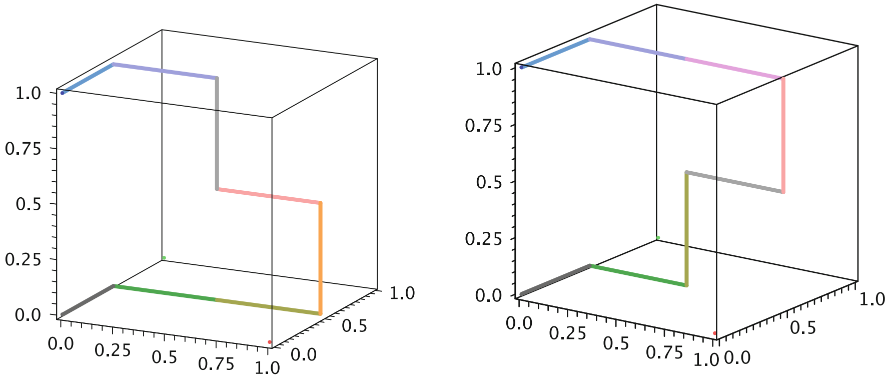


Fig. 8.3 Two further variants from which to choose the approximating polygon of a 3D Hilbert curve

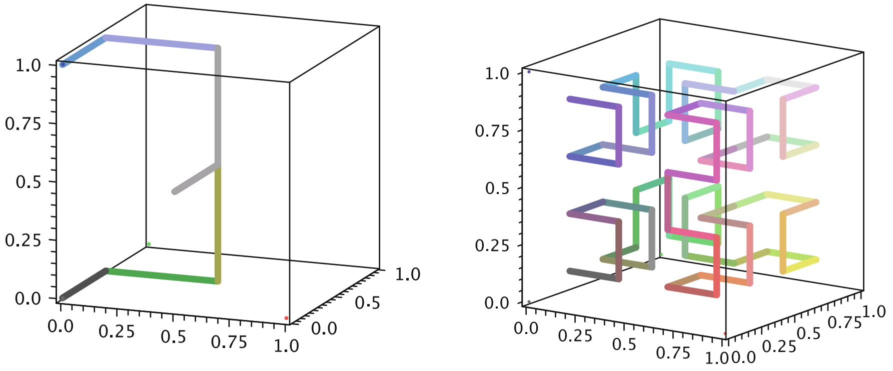


Fig. 8.4 A fourth variant to construct the approximating polygon of a 3D Hilbert curve (together with the respective second iteration)

order. However, the two plots at the top illustrate that the approximating polygons of the two curves differ. Note the different transfers through the third and fourth and through the fifth and sixth subcube. Mixing the two options will thus lead to two further options, which are symmetric to each other – they are given in Fig. 8.3. Finally, a fourth variant to construct the approximating polygon is given in Fig. 8.4. In all cases, the effect of the different choices will only show up in the second iteration of the curves.

Variants Due to Rotations of the Patterns

Figure 8.5 illustrates a further option to construct different variants of the 3D Hilbert curve. There are two possibilities to connect two adjacent corners of the

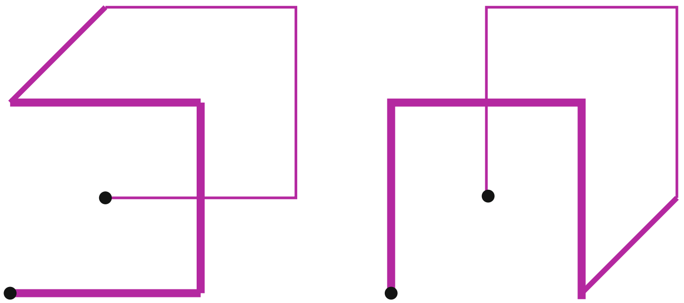


Fig. 8.5 Varying the basic 3D Hilbert pattern by a suitable rotation

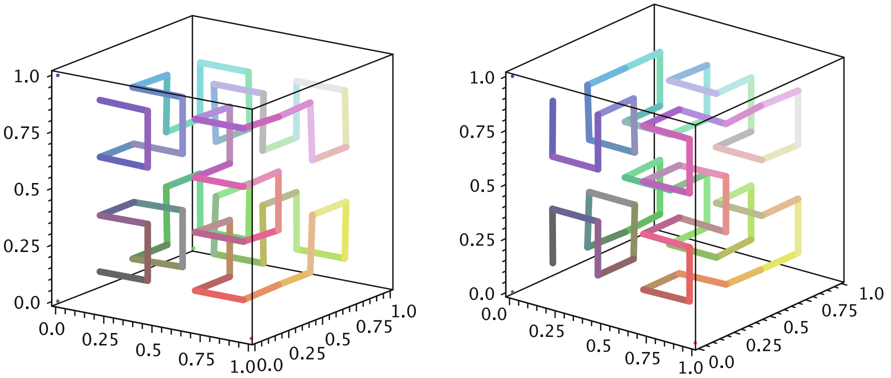


Fig. 8.6 Two different constructions of a 3D Hilbert curve. Note the different orientation of the Hilbert pattern in each subcube despite the identical local start and end points

subcube by a local Hilbert pattern. Hence, for a given approximating polygon, we can choose separately for each of the eight subcubes, which of the variants shown in Fig. 8.5 should be used. Figure 8.6 shows two different 3D Hilbert constructions that demonstrate these options. Note that despite of an identical approximating polygon the orientation of the local Hilbert patterns is different in all of the eight subcubes. Nevertheless, the first iteration is identical, and the eight subcubes of the first level are visited in the same order.

Obviously there are $2^8 = 256$ different variants that result from a different choice of orientation of the local patterns. Adding the four different possibilities to choose the approximating polygon leads to 1,024 potential variants (the two variants in Fig. 8.3 lead to constructions that are pairwise symmetric). As there are also two variants to construct the approximating polygon for the second basic pattern in Fig. 8.1, there are $6 \cdot 2^8 = 1{,}536$ structurally different variants to construct a simple 3D Hilbert curve.

8.1.2 *Arithmetisation of the 3D Hilbert Curve*

The arithmetisation of the 3D Hilbert curve follows the established scheme. Due to the recursive substructuring into eight intervals and cubes in each iteration step, the respective arithmetisation is based on the octal representation of the parameter.

Hence, for a parameter $t = 0_8.k_1k_2k_3k_4\dots$ given as an octal number, we need to determine the operators H_k for the following representation:

$$h(0_8.k_1k_2k_3k_4\dots) = H_{k_1} \circ H_{k_2} \circ H_{k_3} \circ H_{k_4} \circ \cdots \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

Each operator H_0 to H_7 describes the transformation of the unit cube into one of the eight subcubes. For the Hilbert curve illustrated in the left plot of Fig. 8.6, we can derive the following set of operators:

$$H_0\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} \frac{1}{2}y + 0 \\ \frac{1}{2}z + 0 \\ \frac{1}{2}x + 0 \end{pmatrix} \qquad H_1\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} \frac{1}{2}z + 0 \\ \frac{1}{2}x + \frac{1}{2} \\ \frac{1}{2}y + 0 \end{pmatrix}$$

$$H_2\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} \frac{1}{2}z + \frac{1}{2} \\ \frac{1}{2}x + \frac{1}{2} \\ \frac{1}{2}y + 0 \end{pmatrix} \qquad H_3\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} -\frac{1}{2}x + 1 \\ -\frac{1}{2}y + \frac{1}{2} \\ -\frac{1}{2}z + 0 \end{pmatrix}$$

$$H_4\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} -\frac{1}{2}x + 1 \\ -\frac{1}{2}y + \frac{1}{2} \\ \frac{1}{2}z + \frac{1}{2} \end{pmatrix} \qquad H_5\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} -\frac{1}{2}z + 1 \\ \frac{1}{2}x + \frac{1}{2} \\ -\frac{1}{2}y + 1 \end{pmatrix}$$

$$H_6\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} -\frac{1}{2}z + \frac{1}{2} \\ \frac{1}{2}x + \frac{1}{2} \\ -\frac{1}{2}y + 1 \end{pmatrix} \qquad H_7\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} \frac{1}{2}y + 0 \\ -\frac{1}{2}z + \frac{1}{2} \\ -\frac{1}{2}x + 1 \end{pmatrix}$$

Figure 8.7 illustrates how to derive the operators H_0 and H_7: the images of the three canonical unit vectors form the column vectors of the matrix part of the operator, which implements the required rotations and reflections. The translation part of the H_k operators, as for the 2D case, can be read from the starting points of the subcurves in each of the eight octants. Analogous to the 2D Hilbert and Peano arithmetisations, we can use these operators to build algorithms to compute the respective 3D Hilbert mapping, and also the 3D Hilbert index.

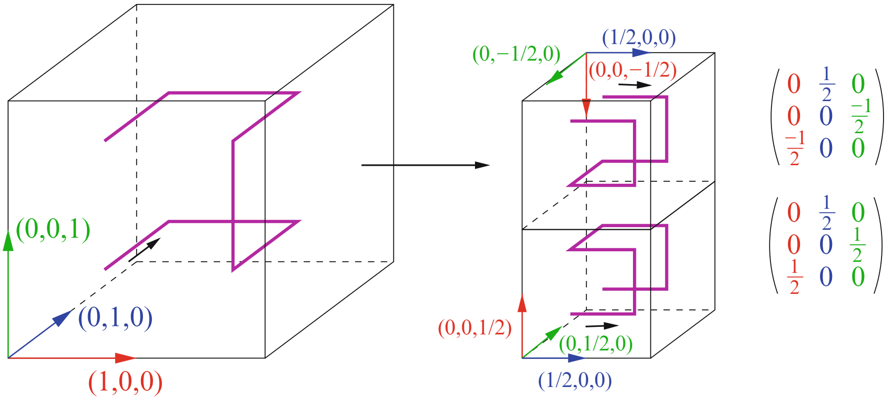

(0,0,1)
(0,1,0)
(1,0,0)
(1/2,0,0)
(0,−1/2,0)
(0,0,−1/2)
(0,0,1/2)
(0,1/2,0)
(1/2,0,0)
$\begin{pmatrix} 0 & \frac{1}{2} & 0 \\ 0 & 0 & \frac{-1}{2} \\ \frac{-1}{2} & 0 & 0 \end{pmatrix}$
$\begin{pmatrix} 0 & \frac{1}{2} & 0 \\ 0 & 0 & \frac{1}{2} \\ \frac{1}{2} & 0 & 0 \end{pmatrix}$

Fig. 8.7 Obtaining the H_k operators for the arithmetisation of a 3D Hilbert curve

8.1.3 A 3D Hilbert Grammar with Minimal Number of Non-Terminals

Grammar representations for 3D Hilbert curves, in general, are tedious, because of the large number of required non-terminal characters. In total, there are 48 different basic patterns: for each of the 12 edges, there is a set of four patterns that have their entry and exit point of the curve adjacent to the given edge – two different patterns are obtained from pattern rotations, as in Fig. 8.6, and two further patterns result from switching the orientation, i.e., exchanging entry and exit point. An exhaustive search on all 1,024 regular 3D Hilbert variants reveals that nearly all grammars for the curves will require 24 non-terminals. However, for each of the four possible approximating polygons, we obtain exactly one curve that requires only 12 non-terminals. The arithmetisation introduced in the previous Sect. 8.1.2, leads to one of these four curves.

The 12 non-terminals for this special 3D Hilbert curve are coded in a specific way to describe the orientation of the corresponding pattern:

$$[xyz], [x\bar{y}\bar{z}], [yzx], [y\bar{z}\bar{x}], [zxy], [z\bar{x}\bar{y}], [\bar{x}y\bar{z}], [\bar{x}\bar{y}z], [\bar{y}z\bar{x}], [\bar{y}\bar{z}x], [\bar{z}x\bar{y}], [\bar{z}\bar{x}y]$$

The triples reflect the image of the general vector (x, y, z) subject to a rotation that converts the starting pattern, $[xyz]$, into the desired pattern. Bars indicate negative values, such that $[x\bar{y}\bar{z}]$ corresponds to the image vector $(x, -y, -z)$, e.g. Thus, the third coordinate direction in each triple reflects the orientation of the edge that is adjacent to the entry and exit points (which is the z direction for the initial pattern $[xyz]$). A bar will consequently indicate that the connection of entry to exit point runs in the negative coordinate direction. The location of x and y in the triple indicate the location of the entry point within the respective plain. A bar indicates that the entry point will be at coordinate 1 in that direction –

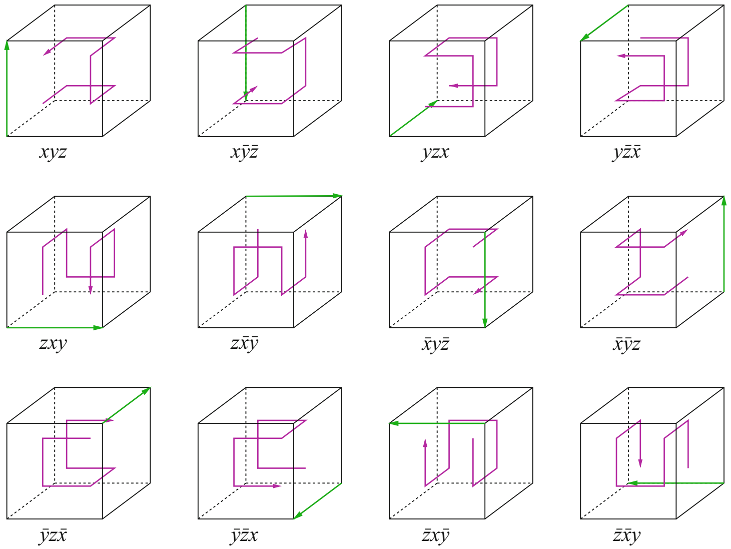


Fig. 8.8 The 12 basic batterns required to construct the 3D Hilbert curve, as introduced in Sect. 8.1.2. The connection of entry and exit point is highlighted by a *green arrow* for each pattern

otherwise at coordinate 0. For example, the pattern $[z\bar{x}\,\bar{y}]$ has its entry point located at coordinates $(0,1,1)$ and the connection to the exit point, at $(1,1,1)$, is in the positive x-direction. All 12 occurring patterns, together with their non-terminals, are illustrated in Fig. 8.8. The 3D Hilbert curve introduced in Sect. 8.1.2 is then described by the following productions:

$$
\begin{aligned}
&[xyz] \longleftarrow [yzx] \times [zxy] \rightarrow [zxy] \cdot [\bar{x}\bar{y}z] \uparrow [\bar{x}\bar{y}z] \times [\bar{z}x\bar{y}] \leftarrow [\bar{z}x\bar{y}] \cdot [y\bar{z}\bar{x}]\\
&[x\bar{y}\bar{z}] \longleftarrow [y\bar{z}\bar{x}] \cdot [z\bar{x}\bar{y}] \rightarrow [z\bar{x}\bar{y}] \times [\bar{x}y\bar{z}] \downarrow [\bar{x}y\bar{z}] \cdot [\bar{z}\bar{x}y] \leftarrow [\bar{z}\bar{x}y] \times [yzx]\\
&[yzx] \longleftarrow [zxy] \rightarrow [xyz] \uparrow [xyz] \leftarrow [\bar{y}z\bar{x}] \times [\bar{y}z\bar{x}] \rightarrow [x\bar{y}\bar{z}] \downarrow [x\bar{y}\bar{z}] \leftarrow [\bar{z}\bar{x}y]\\
&[y\bar{z}\bar{x}] \longleftarrow [z\bar{x}\bar{y}] \rightarrow [x\bar{y}\bar{z}] \downarrow [x\bar{y}\bar{z}] \leftarrow [\bar{y}\bar{z}x] \cdot [\bar{y}\bar{z}x] \rightarrow [xyz] \uparrow [xyz] \leftarrow [\bar{z}x\bar{y}]\\
&[zxy] \longleftarrow [xyz] \uparrow [yzx] \times [yzx] \downarrow [z\bar{x}\bar{y}] \rightarrow [z\bar{x}\bar{y}] \uparrow [\bar{y}\bar{z}x] \cdot [\bar{y}\bar{z}x] \downarrow [\bar{x}y\bar{z}]\\
&[z\bar{x}\bar{y}] \longleftarrow [x\bar{y}\bar{z}] \downarrow [y\bar{z}\bar{x}] \cdot [y\bar{z}\bar{x}] \uparrow [zxy] \rightarrow [zxy] \downarrow [\bar{y}z\bar{x}] \times [\bar{y}z\bar{x}] \uparrow [\bar{x}\bar{y}z]\\
&[\bar{x}y\bar{z}] \longleftarrow [\bar{y}z\bar{x}] \times [\bar{z}x\bar{y}] \leftarrow [\bar{z}x\bar{y}] \cdot [x\bar{y}\bar{z}] \downarrow [x\bar{y}\bar{z}] \times [zxy] \rightarrow [zxy] \cdot [\bar{y}\bar{z}x]\\
&[\bar{x}\bar{y}z] \longleftarrow [\bar{y}\bar{z}x] \cdot [\bar{z}\bar{x}y] \leftarrow [\bar{z}\bar{x}y] \times [xyz] \uparrow [xyz] \cdot [z\bar{x}\bar{y}] \rightarrow [z\bar{x}\bar{y}] \times [\bar{y}z\bar{x}]\\
&[\bar{y}z\bar{x}] \longleftarrow [\bar{z}x\bar{y}] \leftarrow [\bar{x}y\bar{z}] \downarrow [\bar{x}y\bar{z}] \rightarrow [yzx] \times [yzx] \leftarrow [\bar{x}\bar{y}z] \uparrow [\bar{x}\bar{y}z] \rightarrow [z\bar{x}\bar{y}]\\
&[\bar{y}\bar{z}x] \longleftarrow [\bar{z}\bar{x}y] \leftarrow [\bar{x}\bar{y}z] \uparrow [\bar{x}\bar{y}z] \rightarrow [y\bar{z}\bar{x}] \cdot [y\bar{z}\bar{x}] \leftarrow [\bar{x}y\bar{z}] \downarrow [\bar{x}y\bar{z}] \rightarrow [zxy]\\
&[\bar{z}x\bar{y}] \longleftarrow [\bar{x}y\bar{z}] \downarrow [\bar{y}z\bar{x}] \times [\bar{y}z\bar{x}] \uparrow [\bar{z}\bar{x}y] \leftarrow [\bar{z}\bar{x}y] \downarrow [y\bar{z}\bar{x}] \cdot [y\bar{z}\bar{x}] \uparrow [xyz]\\
&[\bar{z}\bar{x}y] \longleftarrow [\bar{x}\bar{y}z] \uparrow [\bar{y}\bar{z}x] \cdot [\bar{y}\bar{z}x] \downarrow [\bar{z}x\bar{y}] \leftarrow [\bar{z}x\bar{y}] \uparrow [yzx] \times [yzx] \downarrow [x\bar{y}\bar{z}]
\end{aligned}
$$

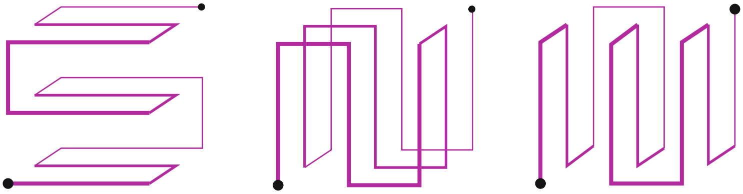


Fig. 8.9 Switch-back patterns for the 3D Peano curve

The set of terminal characters is now $\{\leftarrow, \rightarrow, \downarrow, \uparrow, \cdot, \times\}$, where $\times$ and $\cdot$ represent backward and forward moves in the z direction (imagine the tail or tip of an arrow seen from behind, or from the front, respectively).

8.2 3D Peano Curves

As a 3D Peano curve, we expect a recursive, face-connected space-filling curve constructed from a recursive substructuring of the target domain, the unit cube $[0, 1]^3$, into $3 \times 3 \times 3$ congruent subcubes in each step of the recursion. As there is already a multitude of different Peano curves in the 2D case, and as the example of the Hilbert curve showed us that the degree of freedom to construct a space-filling curve is likely to grow in 3D, we can expect an even larger collection of 3D Peano curves than in 2D. We will therefore concentrate on switch-back curves, only. The respective basic patterns for construction are given in Fig. 8.9. As the approximating polygons of all three patterns are given by the space diagonal, we can again replace these pattern against each other in a construction of a 3D Peano curve, and thus obtain an abundance of different curves.

An interesting property of the 3D Peano patterns that can be observed in Fig. 8.9 is that the patterns are assembled from 2D Peano curves. Moreover, the 3D patterns will traverse the subcubes of any 3×3 plane within the $3 \times 3 \times 3$ subcubes according to a 2D Peano pattern. In this section, we will therefore restrict ourselves to a special type of 3D Peano curves, which can be constructed via such a recursion on the *dimension*.

8.2.1 *A Dimension-Recursive Grammar to Construct a 2D Peano Curve*

Figure 8.10 illustrates that the grammar-based description of the standard 2D Peano curves can be split into two steps that correspond to the horizontal and vertical direction. The respective patterns still can be represented by the usual patterns P, Q, R, and S, which are now characterised by the diagonal relevant for the approximating polygon. However, the respective subdomains of the patterns can be both cubes and rectangles.

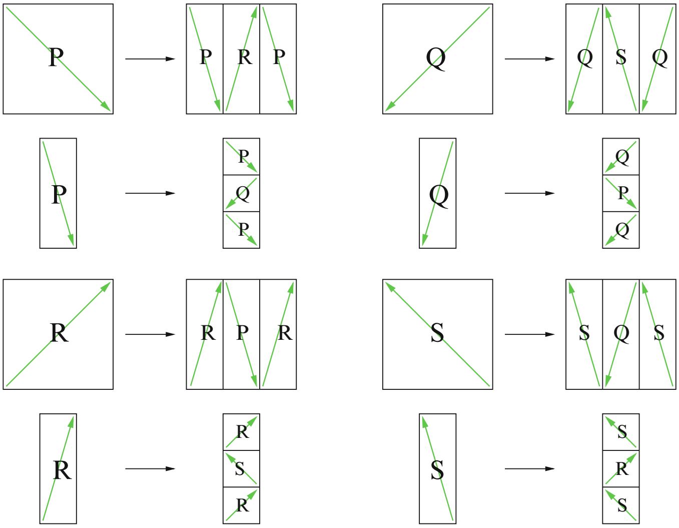


Fig. 8.10 Derivation of the dimension-recursive grammar of a “standard” 2D Peano curve

We can describe this construction by a grammar that extends the set of non-terminals $\{P, Q, R, S\}$ by further non-terminals $\{P_y, Q_y, R_y, S_y\}$ that correspond to the rectangular subdomains. The production rules of the grammar are then:

$$\begin{array}{llll}
P \longleftarrow P_y \rightarrow R_y \rightarrow P_y & \qquad & P_y \longleftarrow P \uparrow Q \uparrow P \\
Q \longleftarrow Q_y \leftarrow S_y \leftarrow Q_y & \qquad & Q_y \longleftarrow Q \uparrow P \uparrow Q \\
R \longleftarrow R_y \rightarrow P_y \rightarrow R_y & \qquad & R_y \longleftarrow R \downarrow S \downarrow R \\
S \longleftarrow S_y \leftarrow Q_y \leftarrow S_y & \qquad & S_y \longleftarrow S \downarrow R \downarrow S
\end{array}$$

The grammar is equivalent to the grammar presented in Sect. 3.3, as can easily be seen by expanding the non-terminals $\{P_y, Q_y, R_y, S_y\}$ in the productions for P, Q, R, and S. We just take an intermediate step via the rectangular subdomains. At this point, take some time to try Exercise 8.4, which deals with the question of how to extend this approach to construct iterations on rectangular meshes and respective Peano curves on rectangular domains – Sect. 8.2.3 will generalise this problem to rectangular grids of (almost) arbitrary size.

8.2.2 *Extension of the Dimension-Recursive Grammar to Construct 3D Peano Curves*

The grammar for the 2D Peano curve can be extended to that for a 3D curve in an almost straightforward way. We now require three sets of non-terminals – the

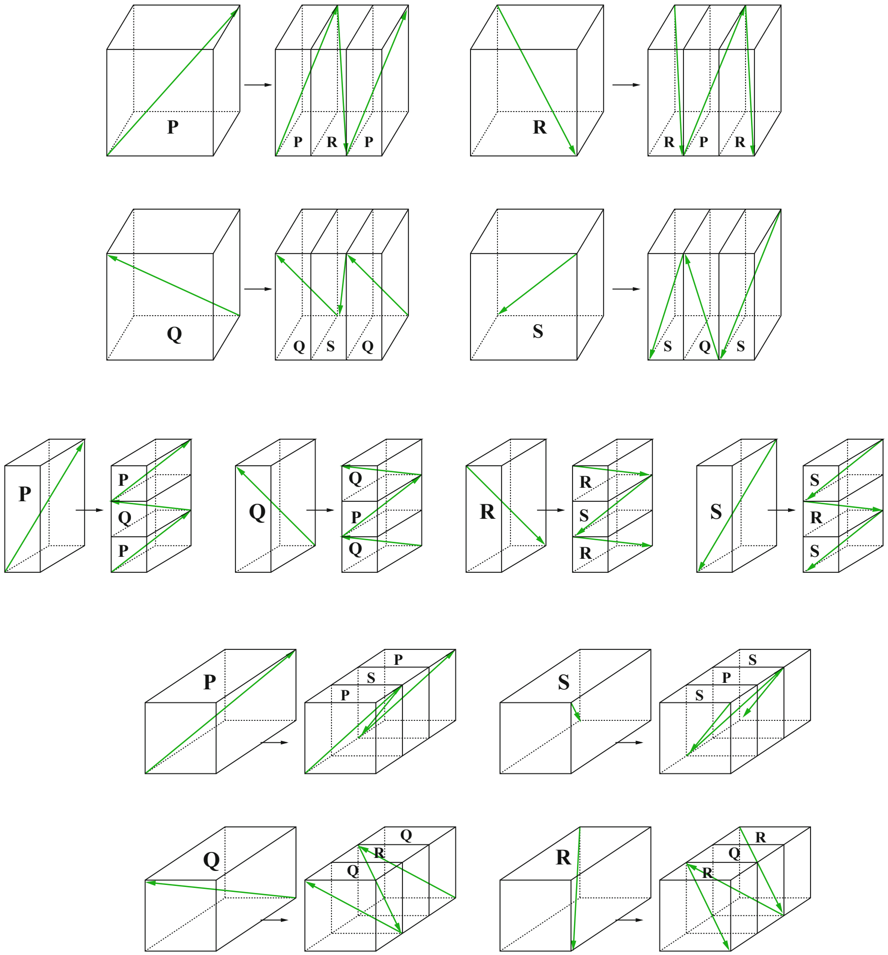


Fig. 8.11 Approximating polygons and corresponding non-terminals to construct a dimension-recursive grammar for a 3D Peano curve (the indices of the non-terminals are omitted, but should be obvious)

non-terminals $\{P, Q, R, S\}$ that represent patterns for the cubic subdomains plus non-terminals $\{P_y, Q_y, R_y, S_y\}$ and $\{P_{yz}, Q_{yz}, R_{yz}, S_{yz}\}$ for the rectangular domains, where we subdivided in only one or two of the three dimension. The convention for the indices of the non-terminals is to take the non-split dimensions (i.e. the longer sides of the rectangle) as indices. The images in Fig. 8.11 illustrate the construction of the 3D Peano curve from these 12 patterns – for illustration purposes, the patterns are depicted by the corresponding approximating polygon, only.

With the terminal characters $\{\leftarrow, \rightarrow, \downarrow, \uparrow, \cdot, \times\}$, we can transfer the 3D Peano curve constructed in Fig. 8.11 into the following grammar:

$$\begin{array}{lll}
P \longleftarrow P_{yz} \rightarrow R_{yz} \rightarrow P_{yz} & P_{yz} \longleftarrow P_y \uparrow Q_y \uparrow P_y & P_y \longleftarrow P \times S \times P \\
Q \longleftarrow Q_{yz} \leftarrow S_{yz} \leftarrow Q_{yz} & Q_{yz} \longleftarrow Q_y \uparrow P_y \uparrow Q_y & Q_y \longleftarrow Q \cdot R \cdot Q \\
R \longleftarrow R_{yz} \rightarrow P_{yz} \rightarrow R_{yz} & R_{yz} \longleftarrow R_y \downarrow S_y \downarrow R_y & R_y \longleftarrow R \cdot Q \cdot R \\
S \longleftarrow S_{yz} \leftarrow Q_{yz} \leftarrow S_{yz} & S_{yz} \longleftarrow S_y \downarrow R_y \downarrow S_y & S_y \longleftarrow S \times P \times S
\end{array}$$

Based on this grammar, Algorithm 8.1 implements a traversal along the iterations of the respective 3D Peano curve. We give the recursive procedure for the non-terminals P, P_y, and P_{yz}. The implementation of the procedures for the remaining non-terminals is straightforward. Note that the reduction of the recursion parameter `depth` is only required in the procedure for P. Hence, the recursion is only stopped on a "cubic" pattern, which also ensures that we perform the same number of substructuring steps in each dimension.

Algorithm 8.1: 3D Peano traversal

```
Procedure P(depth) begin
    if depth > 0 then
        Pyz(depth-1); right();
        Ryz(depth-1); right();
        Pyz(depth-1);
    end
end
Procedure Pyz(depth) begin
    Py(depth); up();
    Qy(depth); up();
    Py(depth);
end
Procedure Py(depth) begin
    P(depth); back();
    S(depth); back();
    P(depth);
end
```

8.2.3 *Peano Curves Based on* 5 × 5 *or* 7 × 7 *Refinement*

The 3×3 switchback pattern to construct the regular Peano curves is easily extended to a 5×5 or 7×7 pattern, as illustrated in Fig. 8.12. In fact, we can define such a pattern on any $n \times m$ grid, provided n and m are odd numbers. An extension to the 3D case is possible in the same way, and by recursive extension, we will obtain Peano iterations on $n^k \times m^k$ (in 2D) or $n^k \times m^k \times l^k$ (in 3D) cells. For $k \rightarrow \infty$, we thus obtain a new family of Peano curves.

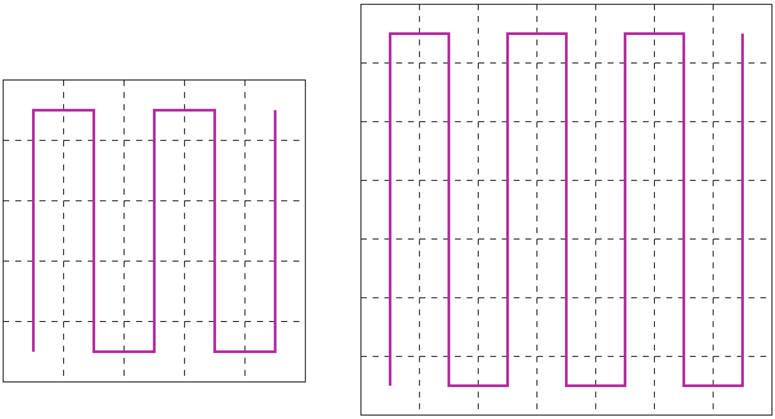


Fig. 8.12 First iteration of a Peano curve based on a 5 × 5 or 7 × 7 switchback pattern

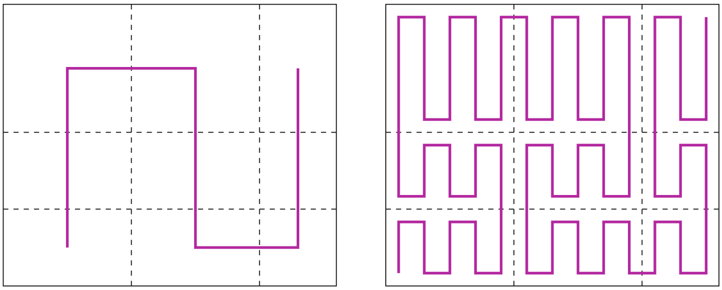


Fig. 8.13 Second iteration of a Peano curve on a $(5 + 5 + 3) \times (5 + 3 + 3)$ grid

Peano Iterations on $(2n + 1) \times (2m + 1)$ or $(2n + 1) \times (2m + 1) \times (2l + 1)$ Grids

Peano curves based on 5 × 5 or 7 × 7 refinement are not too often used; however, as we will discuss in this section, we can apply the respective patterns to generate Peano iterations on grids of (nearly) arbitrary size. We will stick to the regular 3 × 3 recursion, and only allow other patterns on the leaf level. A simple example is given in Fig. 8.13.

In a 3×3 recursion, we have to split the number of cells in each of the dimensions into a sum of three numbers that determine the size of the subsquares. Hence, we exploit the fact that any odd number can be written as a sum of three odd numbers of nearly equal size:

$$2n+1 = \begin{cases} k+k+k & \text{if } 2n+1 = 3k, \\ k+k+(k+2) & \text{if } 2n+1 = 3k+2, \\ k+(k+2)+(k+2) & \text{if } 2n+1 = 3k+4, \end{cases} \tag{8.1}$$

Algorithm 8.2: 3D Peano traversal on an $l \times m \times n$ grid (l, m, n odd)

```
Procedure P(l,m,n) begin
    if $l < 9$ and $m < 9$ and $n < 9$ then
        leafP(l,m,n);
        return
    end
    if $l >= m$ and $l >= n$ then
        P(split (l,1), m, n); right();
        R(split (l,2), m, n); right();
        P(split (l,3), m, n);
        return
    end
    if $l < m$ and $m >= n$ then
        P(l, split (m,1), n); up();
        Q(l, split (m,2), n); up();
        P(l, split (m,3), n);
        return
    end
    if $l < n$ and $m < n$ then
        P(l, m, split (n,1)); back();
        S(l, m, split (n,2)); back();
        P(l, m, split (n,3));
        return
    end
end
```

where we assume that k is again an odd number. Note that if we split two subsequent odd numbers, $2n+1$ and $2n+3$, into such a sum, the summands will still only differ by 2, at most:

$$2n+3 = \begin{cases} k + k + (k+2) & \text{if } 2n+1 = 3k, \\ k + (k+2) + (k+2) & \text{if } 2n+1 = 3k+2, \\ (k+2) + (k+2) + (k+2) & \text{if } 2n+1 = 3k+4. \end{cases} \tag{8.2}$$

Further splits of k and $k+2$ will again lead to sums of three odd numbers that differ by 2, at most. By induction, we can thus prove that any odd number can be split into a sum of 3^p odd numbers, which differ by at most 2. Hence, we can generalise the simple pattern given in Fig. 8.13 for any grid of size $(2n+1)\times(2m+1)$. Obviously, the leaf patterns will have sizes $m \times n$, where m and n may be 3, 5, or 7, because for $m, n = 9$, there would be a further step of recursion.

We can thus generate Peano iterations on arbitrary grid sizes, provided the grid sizes are given by odd numbers. However, even if data structures have even dimension sizes, padding by one additional element in an even-sized dimension will allow us to use such Peano iterations to generate element numberings on these data structures. Algorithm 8.2 sketches an implementation to produce such Peano iterations. The function `split` is used to compute the sizes of the split subsquares according to Eqs. (8.1) and (8.2). The leaf-case in Algorithm 8.2 needs to be implemented in the function `leafP` (for the pattern P). Note that we can call the leaf case for larger sizes of the leaf squares, as well. We just need to change the respective stopping criterion in the recursion, and replace 9 by the desired maximum leaf size.

8.2.4 Towards Peano's Original Construction

The dimension-recursive construction of the 2D Peano curve can also be represented via an arithmetisation:

$$p(0_3.t_1t_2t_3t_4\ldots) = P^x_{t_1} \circ P^y_{t_2} \circ P^x_{t_3} \circ P^y_{t_4} \circ \cdots \begin{pmatrix} 0 \\ 0 \end{pmatrix}.$$

The alternating substructuring along the different dimensions is reflected by the two groups of operators P^x_j and P^y_j, and the subdivision into three sub-rectangles in each step leads to the construction via the ternary system. The operators P^x_j are easily derived as

$$P^x_0 \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x+0 \\ \frac{1}{3}y+0 \end{pmatrix} \quad P^x_1 \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} -x+1 \\ \frac{1}{3}y+\frac{1}{3} \end{pmatrix} \quad P^x_2 \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x+0 \\ \frac{1}{3}y+\frac{2}{3} \end{pmatrix}.$$

In its recursive formulation, the arithmetisation of the dimension-recursive Peano curve can then be written as

$$p(t) = p(0_3.t_1t_2t_3\ldots) = P^x_{t_1} \circ \begin{pmatrix} 0_3.x_1x_2\ldots \\ 0_3.y_1y_2\ldots \end{pmatrix} = P^x_{t_1} \circ p(0_3.t_2t_3\ldots) = P^x_{t_1} \circ p(\tilde{t}).$$

For $t_1 = 0$ and $t_1 = 2$, the bit representation of $p(t)$ stays simple,

$$P^x_0 \circ \begin{pmatrix} 0_3.x_1x_2\ldots \\ 0_3.y_1y_2\ldots \end{pmatrix} = \begin{pmatrix} 0_3.x_1x_2\ldots \\ 0_3.0y_1y_2\ldots \end{pmatrix} \quad P^x_2 \circ \begin{pmatrix} 0_3.x_1x_2\ldots \\ 0_3.y_1y_2\ldots \end{pmatrix} = \begin{pmatrix} 0_3.x_1x_2\ldots \\ 0_3.2y_1y_2\ldots \end{pmatrix},$$

however, for $t_1 = 1$, all ternary digits of the x-coordinate are reversed:

$$P^x_1 \circ \begin{pmatrix} 0_3.x_1x_2\ldots \\ 0_3.y_1y_2\ldots \end{pmatrix} = \begin{pmatrix} 1 - 0_3.x_1x_2\ldots \\ 0_3.1y_1y_2\ldots \end{pmatrix} = \begin{pmatrix} 0_3.(2-x_1)(2-x_2)\ldots \\ 0_3.1y_1y_2\ldots \end{pmatrix}.$$

Any subsequent occurrence of a 1-digit will lead to a further reversal of digits. The number of reversals is thus given by the number of preceding 1-digits. For the P^y operators, we observe the same behaviour – with the exception that the digits in the y-coordinate are reversed. In his original construction [214], Giuseppe Peano used such reversal of ternary digits to describe his space-filling curve.

Definition 8.1 (Peano curve, Peano's original construction). Consider a parameter $t \in \mathcal{I} := [0,1]$ given in ternary representation:

$$t = 0_3.t_1 t_2 t_3 t_4 \dots$$

The Peano mapping $p\colon \mathcal{I} \to \mathcal{Q} := [0,1] \times [0,1]$ is then defined as

$$p(t) := \begin{pmatrix} 0_3.t_1\, k^{t_2}(t_3)\, k^{t_2+t_4}(t_5) \dots \\ 0_3.k^{t_1}(t_2)\, k^{t_1+t_3}(t_4) \dots \end{pmatrix},$$

where the operator k is defined as $k(t_i) := 2 - t_i$ for $t_i = 0, 1, 2$, and k^j will apply the operator k exactly j times.

The operator k has the property

$$k^2(t) = t \quad \Rightarrow \quad k^{2n}(t) = t \quad \text{for all } n;$$

hence, only 1-digits will lead to a reversal of digits. Note that the ternary representation must not be stopped after a finite number of digits – otherwise a wrong result will be obtained:

$$p(0_3.11) \overset{?}{=} \begin{pmatrix} 0_3.1 \\ 0_3.k^1(1) \end{pmatrix} = \begin{pmatrix} 0_3.1 \\ 0_3.1 \end{pmatrix} = \begin{pmatrix} \frac{1}{3} \\ \frac{1}{3} \end{pmatrix}$$

$$p(0_3.1100) \overset{?}{=} \begin{pmatrix} 0_3.1k^1(0) \\ 0_3.k^1(1)k^1(0) \end{pmatrix} = \begin{pmatrix} 0_3.12 \\ 0_3.12 \end{pmatrix} = \begin{pmatrix} \frac{5}{9} \\ \frac{5}{9} \end{pmatrix}$$

$$p(0_3.110000\dots) \overset{!}{=} \begin{pmatrix} 0_3.1k^1(0)k^1(0)\dots \\ 0_3.k^1(1)k^1(0)k^1(0)\dots \end{pmatrix} = \begin{pmatrix} 0_3.122\dots \\ 0_3.122\dots \end{pmatrix}$$
$$= \begin{pmatrix} 0_3.2 \\ 0_3.2 \end{pmatrix} = \begin{pmatrix} \frac{2}{3} \\ \frac{2}{3} \end{pmatrix}.$$

As trailing 0 digits will be reversed, as well, we always have to compute $p(t)$ using an infinite ternary representation of t.

8.3 A 3D Sierpinski Curve

The Sierpinski curve can be extended from the 2D to the 3D case, as well. One of the main difficulties, however, is the recursive substructuring of the target domain. For the 3D Hilbert and Peano curve the step from squares to cubes was intuitive, and the recursive subdivision of each cube into 8 or 27 subcubes respectively was straightforward. For a 3D Sierpinski curve, we expect a tetrahedral target domain, instead. The question of how a tetrahedron is recursively substructured into smaller

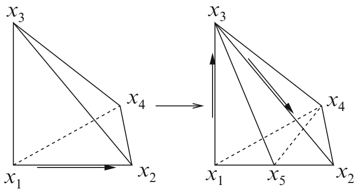


Fig. 8.14 Bisection of a tetrahedron with tagged edge. The oriented tagged edges are marked by respective *arrows*

tetrahedra, however, is non-trivial, and we will come back to this question in Chap. 12. For the following construction, we will sacrifice the feature of having a substructuring into congruent subdomains. To obtain such a substructuring, we generalise the bisection approach with tagged edges that was used in Sect. 6.2.

Definition 8.2 (Tetrahedron with Tagged Edge). Given are the four corner points $x_1, x_2, x_3, x_4 \in \mathbb{R}^3$ of a tetrahedron. The quad-tuple

$$[x_1, x_2, x_3, x_4]$$

shall then be defined as the *tetrahedron with tagged edge* x_1x_2, where the edge x_1x_2 is assumed to be oriented.

Similar to the bisection of triangles, we can split the given tetrahedron along the tagged edge, and obtain two sub-tetrahedra:

$$[x_1, x_2, x_3, x_4] \to [x_1, x_3, x_4, x_5]\,, [x_3, x_2, x_4, x_5]\,. \tag{8.3}$$

The new corner x_5 is, by default, chosen as the midpoint of the tagged edge x_1x_2. Figure 8.14 shows an example of such a bisection. The tagged edges are again oriented such that they can replace the previous tagged edge by a respective polygon. Hence, as in the 2D case, the tagged edges will serve as the approximating polygon of a 3D Sierpinski curve. Encouraged by this analogy, we can formulate the respective definition of the 3D Sierpinski curve:

Definition 8.3 (3D Sierpinski Curve). Let $\mathcal{I} := [0,1]$, and $\mathcal{S} = [x_1, x_2, x_3, x_4]$ be a tetrahedron with tagged edge x_1x_2. The mapping function $s\colon \mathcal{I} \to \mathcal{S}$ shall then be defined via the following instructions:

- For each parameter $t \in \mathcal{I}$, there is a sequence of nested intervals

  $$\mathcal{I} \supset [a_1, b_1] \supset \ldots \supset [a_n, b_n] \supset \ldots,$$

  where each interval results from bisecting the parent interval: $[a_k, b_k] = [i_k \cdot 2^{-k}, (i_k+1)2^{-k}], i_k = 0, 1, 2, \ldots, 2^k - 1$.
- Each sequence of intervals corresponds to a sequence of tetrahedra generated by bisection according to Eq. (8.3), S being the initial tetrahedron.
- The resulting sequence of 3D-tetrahedra converges uniquely to a point in $\mathcal{S}$ – this point shall be defined as $s(t)$.

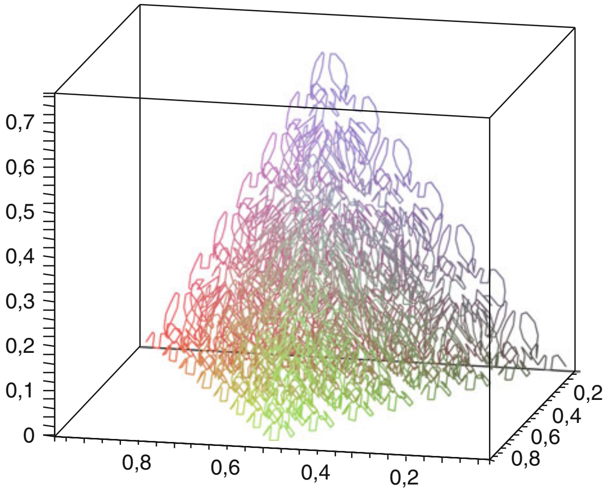


Fig. 8.15 3D Sierpinski curve on an equilateral tetrahedron (12-th iteration)

The image of the mapping $s : \mathcal{I} \to \mathcal{S}$ then defines a *3D Sierpinski curve*.

Analogous to Algorithm 6.1 for the generalised Sierpinski curve, we can turn Definition 8.3 into vertex-labelling Algorithm 8.3 to compute the iterations of the 3D Sierpinski curve. Figure 8.15 plots an iteration of the 3D space-filling curve generated by this algorithm. An equilateral tetrahedron was used as target domain.

Algorithm 8.3: Computing the n-th iteration of a 3D Sierpinski curve

```
Procedure sierp3D(x1, x2, x3, x4, n)
    Parameter: x1,x2,x3,x4: vertices (as 3D coordinates); n: refinement level
    Data: curve: list of vertices (empty at start, contains iteration points on exit)
begin
    if n = 0 then
        return attach(curve, center(x1, x2, x3, x4))
    else
        sierp3D(x1, x3, x4, midpoint(x1,x2), n-1);
        sierp3D(x3, x2, x4, midpoint(x1,x2), n-1);
    end
end
```

As we will see in Chap. 12, the construction of tetrahedra for the presented 3D Sierpinski curve leads to distorted tetrahedra and thus to a space-filling curve with inferior locality properties However, it will also turn out that the given curve is the only possibility to construct a face-connected 3D Sierpinski curve.

References and Further Readings

The first 3D Hilbert curves were apparently described by Bially [42], who provided a state diagram to compute the respective mappings (see Sect. 4.6.2), and by Butz [58], who provided an algorithm for an n-dimensional curve. Bially's curve, similar to the curve described in Sect. 8.1.3, came by with only 12 patterns (which are equivalent to the number of states in his state diagram) – Fig. 8.4 plots its first iteration and approximating polygon.

The 3D Hilbert curve in Fig. 8.2 (left plots) was found independently by Sagan [232, 233]. The curve in the right plots of Fig. 8.2 was presented by Gilbert [99]. Alber and Niedermeier [9] discussed the number of possible variants of 3D Hilbert curves, and also provide a generalisation of the Hilbert curve to construct higher-dimensional curves. Further algorithms for higher-dimensional Hilbert curves were introduced by Quinqueton and Berthod [224] and Kamata et al. [145]. Hamilton and Rau-Chaplin [118] discussed *compact* Hilbert indexings for higher-dimensional spaces with the added requirement that different dimensions may demand a considerably different number of indices in that direction. The respective indexings are kept compatible to the standard curves, though. The credit for the first higher-dimensional curves, however, is likely to go to Bially [42], who provided a state diagram for a 4D curve, and to Butz [58] (both in 1969) – at least, they were among the first to provide respective algorithms.

Peano, in his original paper [214], had already discussed the 3D version of his curve. Also, he had pointed out that the curve was trivial to be adapted to constructions based on any odd number, such as in Sect. 8.2.3. We will take up the discussion of 3D Sierpinski curves in Chap. 12; hence, see the references section of this chapter for background on 3D Sierpinski curves. The simplest option to construct a 3D (or even higher-dimensional) space-filling order is probably to use Morton order – a respective bit-oriented mapping is straightforward for any given dimension. Exercise 8.6 deals with the question how the parameter interval should be chosen to obtain a continuous mapping.

What's next?

- The remainder of this book will focus on applications of space-filling curves. The next chapter will start with the combination of quadttrees and octrees with space-filling curves. Chapter 10 will focus on using space-filling curves as tools for parallelisation. It is recommended to do these two chapters as a block.
- The question of 3D Sierpinski curves is much more complicated than our first discussion in this chapter. For applications in Scientific Computing, it is especially connected with the question how to construct tetrahedral meshes. Chapter 12 will discuss this question in more detail.

Exercises

8.1. In Fig. 8.2, the approximating polygon and iteration in the two left plots illustrate the 3D Hilbert curve as suggested by Sagan [232, 233]. Derive the operators $H_0, \dots, H_7$ for the arithmetisation of this 3D Hilbert curve.

8.2. Try to construct a closed 3D curve of Hilbert-type, analogous to the 2D Hilbert-Moore curve. Give a respective arithmetisation.

8.3. Construct a Hilbert-type curve from the central basic pattern in Fig. 8.1. Again, give the respective arithmetisation.

8.4. Describe how to extend the dimension-recursive grammar for the 2D Peano curve to generate iterations that traverse a rectangular grid of $3^m \times 3^n$ grid cells. Formulate a respective traversal algorithm. You should also discuss how the respective modified iterations could lead to the definition of a modified Peano curve on rectangular domains.

8.5. Check whether the 3D Sierpinski construction is Sect. 8.3 can be extended to higher-dimensional curves. In particular, check whether the bisection rule

$$[x_1, x_2, x_3, x_4, x_5] \to [x_1, x_3, x_4, x_5, x_m], [x_3, x_2, x_4, x_5, x_m],$$

with $x_m = \frac{1}{2}(x_1 + x_2)$ leads to a sequence of 4-simplices that share a common 3-dimensional simplex as hyperface.

8.6. In 2D, we need to choose the Cantor Set as parameter interval to turn Morton order into a continuous mapping (the Lebesgue curve). Discuss how the parameter set should look like in the 3D case. (Is it still possible to use the Cantor Set?)

Chapter 9
Refinement Trees and Space-Filling Curves

In Sect. 1.1, we introduced quadtrees and corresponding refinement trees, as shown in Fig. 9.1, as efficient data structures for geometric modelling. We have even considered a sequential order on the respective grid cells, which we now recognise as equivalent to a Hilbert curve. In this chapter, we will add the respective details to this idea – we will discuss a highly memory-efficient bitstream data structure to store quadtree and octree structures, extend the idea to general spacetrees, and, above all, introduce how to generate and compute sequential orderings of the grid cells based on space-filling curves.

9.1 Spacetrees and Refinement Trees

Quadtrees and octrees subdivide the side length of the squares and cubes into two parts in each step. However, respective spatial tree-oriented structures can easily be generalised to allow substructuring into three or four parts in each step. Figure 9.2 provides an example for a 3 × 3 partitioning in each step, and also shows a corresponding adaptive Peano curve. We could call the respective grid a nonal-tree grid (or nontree grid?!), but for a corresponding 3D grid (and its refinement in 27 subcells in each step), we finally have to look for a better naming scheme.

Definition 9.1 (k^d-spacetree). A (possibly attributed[1]) tree is called a k^d-spacetree, if it has the following properties:

- Each node of the tree is either a leaf, or has exactly k^d children. Each child node is again a k^d-spacetree.
- Each node represents a d-dimensional hypercube. In particular, this holds for each leaf node, as well.

---

[1] We call a tree *attributed*, if each node or leaf stores additional information, as for example inside-outside information, material properties, or similar.

M. Bader, *Space-Filling Curves*, Texts in Computational Science and Engineering 9,
DOI 10.1007/978-3-642-31046-1_9, © Springer-Verlag Berlin Heidelberg 2013

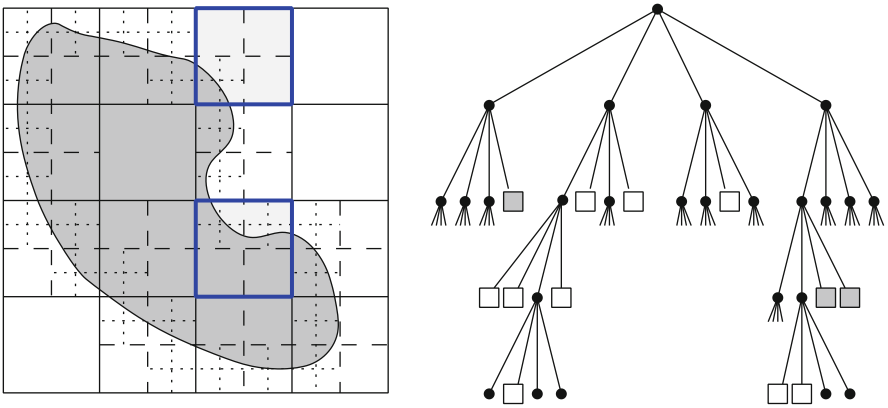


Fig. 9.1 Generating the quadtree representation of a 2D domain. The quadtree is fully extended only for the highlighted cells (see also Fig. 1.2)

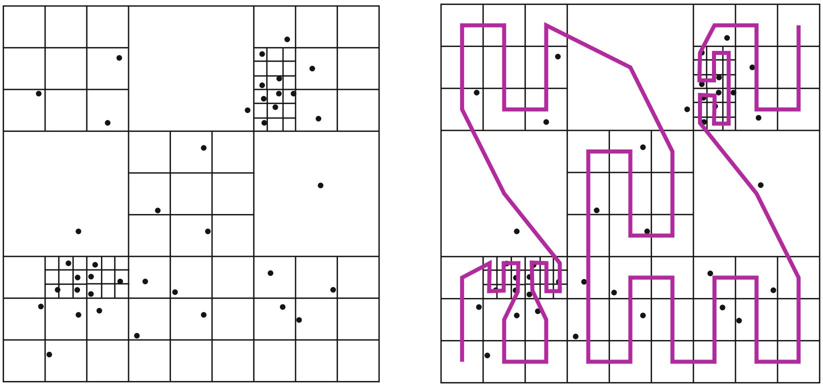


Fig. 9.2 An adaptive spacetree grid with 3 × 3 partition in each refinement step. The grid refinement is triggered by a set of particles, and the refinement steps are chosen such that each leaf-level grid cell finally contains at most one particle

- From each tree level to the next, the side lengths of the hypercubes are reduced by a factor of k^{-1}.

A spacetree is called *regularly refined*, if all leaf nodes are on the same refinement level. Otherwise, it is called an *adaptive* spacetree.

According to this definition, a quadtree is a 2^2-spacetree, and an octree corresponds to a 2^3-spacetree. While we will restrict the term spacetree to grids that have cubes or hypercubes as cells, it is clear that the refinement principle is easily extended to cuboid cells. We will also extend the principle to triangular and tetrahedral grids.

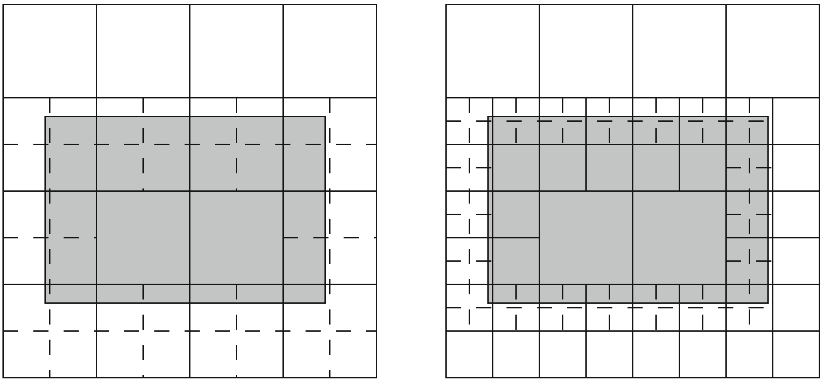


Fig. 9.3 Quadtree refinement for a rectangle that is parallel to the coordinate axes

9.1.1 Number of Grid Cells for the Norm Cell Scheme and for a Quadtree

Intuitively, we expect that a quadtree will require significantly fewer cells to describe a given object with the same resolution as the norm cell scheme. After all, we may use much coarser cells in the interior and outside of the object. However, can this advantage be quantified?

We will try to determine the different number of cells for a simple, 2D example: we prescribe a rectangle that is parallel to the coordinate axes, has side lengths $\frac{3}{4}$ and $\frac{1}{2}$, but is placed such that it will never exactly match the refined subcells (i.e., the coordinates are not finite binary fractions). The object will be embedded into the unit square $[0, 1]^2$, which forms the root element of our quadtree refinement. The first quadtree refinement steps are illustrated in Fig. 9.3. For that rectangle, we can exactly determine how many octree elements are required, if the refinement is increased step by step – the respective number of elements is then compared with the norm cell scheme.

Note that in the norm cell scheme, all cells of the (uniform) grid will be replaced by four cells of half the size. Hence, independent of the size of the rectangle, we will require n^2 cells, if our cells have side lengths of $h = n^{-1}$. The number of cells thus grows of order $\mathcal{O}(h^{-2})$ with the resolution h.

In the quadtree grid, we will only refine cells that contain the boundary of the rectangle. Assume that a quadtree has already been refined up to a cell size $2h$ and contains r cells on the boundary and s cells that are entirely inside or outside of the rectangle. Each of the r boundary cells will be replaced by 4 cells of size h. However, only $2r$ of these $4r$ cells will again be on the boundary: along the boundary, it is obvious that for each cell exactly 2 of 4 smaller cells are again on the boundary. The four corner cells will be replaced by 16 cells of half the size – again, 8 of them will again be on the boundary (compare Fig. 9.3 and note that the side lengths were chosen as multiples of the cell size). Hence, the refined quadtree of cell size h will have $2r$ boundary cells plus $s + 2r$ cells that will not be further refined. We therefore obtain the following recurrence equation for the number of grid cells s_k and r_k of a quadtree of refinement level k:

$$r_k = 2r_{k-1}, \qquad s_k = s_{k-1} + 2r_{k-1}.$$

Note that this recurrence does not hold for the first two steps, $k = 0$ and $k = 1$, where the grid only consists of the root cell or four boundary cells. In refinement level $k = 2$, as denoted by the solid lines in the left image of Fig. 9.3, we have $r_2 = 10$ boundary cells and $s_2 = 6$ interior or exterior cells. The recurrence then holds for all $k \geq 3$, and leads to the following formula for the number of boundary cells:

$$r_k = 2 \cdot r_{k-1} = \ldots = \underbrace{2 \cdot 2 \cdot \ldots \cdot 2}_{(k-2)\text{ times}} \cdot r_2 = 2^{k-2} \cdot 10 = \frac{5}{2} \cdot 2^k.$$

For the sum s_k of interior and exterior cells, we therefore obtain

$$\begin{aligned} s_k &= s_{k-1} + 2r_{k-1} = s_{k-2} + 2r_{k-2} + 2r_{k-1} = \ldots = s_2 + 2\sum_{\kappa=2}^{k-1} r_\kappa \\ &= 6 + 2 \cdot \frac{5}{2} \sum_{\kappa=2}^{k-1} 2^\kappa = 6 + 5(2^k - 1 - 2^1 - 2^0) = 5 \cdot 2^k - 14. \end{aligned}$$

A quadtree of level k therefore consists of $r_k + s_k = \frac{15}{2} \cdot 2^k - 14 \approx \frac{15}{2} \cdot 2^k$ cells. In comparison with the size $h = 2^{-k}$ of a grid cell, the quadtree therefore consists of approximately $\frac{15}{2} \cdot h^{-1}$ grid cells. Hence, while we have $\mathcal{O}(h^{-2})$ grid cells in the norm cell scheme, the octree requires only $\mathcal{O}(h^{-1})$ cells.

In general, the number of required cells no longer grows with the area of the modelled domain, but depends on the length of its boundary – note that $\frac{5}{2}$ is exactly the length of the boundary of our rectangle. In 3D, the number of grid cells will depend on the area of the surface, instead of the domain's volume. A detailed discussion including proofs for this property was given by Faloutsos [85]. However, this considerable reduction only occurs, if the boundary is sufficiently smooth – as a counter-example, imagine a domain that is bounded by the Hilbert-Moore curve!

9.2 Using Space-Filling Curves to Sequentialise Spacetree Grids

In the introduction, we have introduced the problem of sequentialisation of multi-dimensional data structures. For any data structure, its sequentialisation is an important issue, as it can serve as memory layout, element traversal, or just to

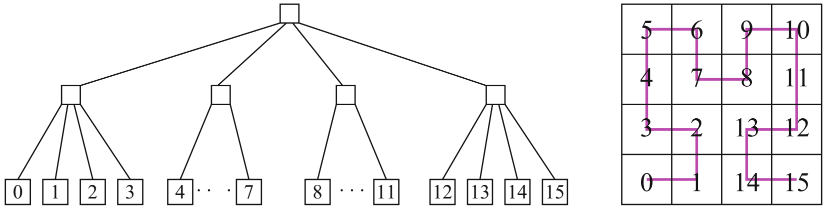


Fig. 9.4 Quadtree representation of a uniformly refined grid, and its sequentialisation by a Hilbert curve iteration

make problems simpler via the induced sequential order. Our sequential approach, naturally, will be based on space-filling curves.

Similar to spacetrees, Hilbert and Peano curves result from a recursive substructuring process. The 2D Hilbert curve is obviously constructed from a (regularly refined) 2^2-spacetree (quadtree); similarly, the construction of 3D Peano curves is based on a 3^3-spacetree. We can therefore sequentialise the leaf cells of a regularly refined 2^2-spacetree (of depth n) by simply following the n-th iteration of the Hilbert curve (compare the example in Fig. 9.4). In the same way, iterations of the Peano curve can be used to sequentialise regularly refined 3^d-spacetrees.

To compute the sequentialisation, we can use the traversal algorithms as introduced in Chap. 3. For example, Algorithm 3.1 will compute the sequentialisation of the tree in Fig. 9.4. Remember that this traversal algorithm was derived from the grammar-based description of the Hilbert curve, where the productions

$$\begin{aligned}
H &\longleftarrow A \uparrow H \rightarrow H \downarrow B \\
A &\longleftarrow H \rightarrow A \uparrow A \leftarrow C \\
B &\longleftarrow C \leftarrow B \downarrow B \rightarrow H \\
C &\longleftarrow B \downarrow C \leftarrow C \uparrow A
\end{aligned}$$

define the recursive scheme of the algorithm: Each of the non-terminal characters H, A, B, and C is turned into a recursive procedure, and the productions determine the (nested) recursive calling scheme; the quadtree then has exactly the same structure as the call tree of the recursive algorithm – compare Fig. 3.3 on page 34 and see the annotated quadtree in Fig. 9.5. The sequential order results from a *depth-first* traversal of this call tree, i.e. of the quadtree:

- If possible, we descend in the tree, always choosing the leftmost node that has not been visited before.
- On leaf level, we process the current element, for example attach its number in the sequential order, store or retrieve from the corresponding memory location, update the node values, or similar.
- Afterwards, we backtrack upwards in the tree to the first (i.e. lowest-level) node that has not been visited, yet, and proceed with the traversal.

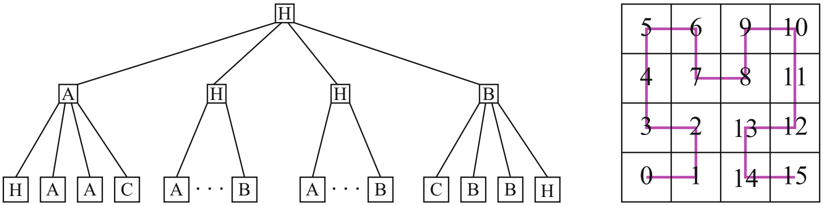


Fig. 9.5 Recursive call tree for the Hilbert traversal algorithm

Hence, we always visit the nodes in the tree representation strictly from left to right. For the corresponding grid cells, the algorithm determines a fixed sequence as well, which directly results from the grammar for the underlying space-filling curve. For the non-terminal H, for example, the sequence starts in a lower-left corner, and then proceeds with the steps up–right–down.

The geometrical position of each grid cell is therefore uniquely determined by the position of its corresponding node in the quadtree representation. Hence, the depth of the quadtree is the only data, which we have to store to be able to retrieve the grid structure, including the position of all grid cells (provided, of course, we store the size of the root cell of the tree). In this sense, the quadtree structure does not require more memory than to store a uniform grid in a classical, array-oriented way, where we would only store the number of grid points per dimension plus the size of the grid cells.

9.2.1 Adaptively Refined Spacetrees

When introducing the spacetree structure, our intention was of course not to be limited by the uniform refinement any more. Hence, we will extend our approach to be able to sequentialise adaptively refined spacetrees in the same way:

- We keep using a *depth-first* traversal of the quadtree.
- We stick to the left-to-right sequence for processing the nodes of the tree, and thus retain the local ordering of the spacetree cells in each refinement step, which is given by the grammar of the space-filling curve.
- However, we can no longer use the depth of the tree to determine whether a node is a leaf. Instead, we add a respective “refinement bit” to each node of the tree, which tells us whether the respective node corresponds to a leaf cell or to a cell that is refined.

Algorithm 9.1 outlines the scheme of such an adaptive tree traversal (compared to the non-adaptive traversal in Algorithm 3.2, only the if-condition is changed). Figure 9.6 illustrates the generated traversal for a simple adaptive spacetree grid. However, the implementation outlined in Algorithm 9.1 is obviously not able to

Algorithm 9.1: Algorithmic sketch of an adaptive Hilbert traversal

```
Procedure H() begin
    if node cell is a leaf then
        // Execute task on current position
        execute (...);
    else
        A(); up();
        H(); right();
        H(); down();
        B();
    end
end
```

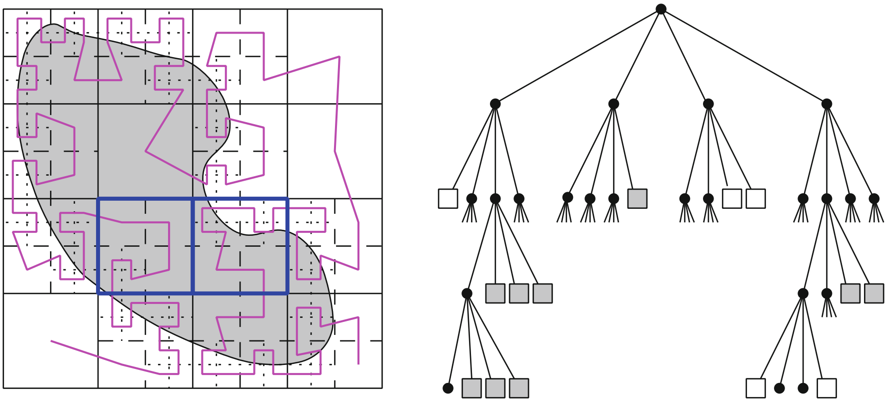


Fig. 9.6 An adaptive spacetree grid and its sequentialisation according to the Hilbert curve

produce the non-aligned steps that are required between quadtree cells of different size. The necessary additions can be required by considering an improved grammar representation for the Hilbert order.

9.2.2 A Grammar for Adaptive Hilbert Orders

To turn a classical Hilbert grammar, for example

$$
\begin{array}{lcl}
H \longleftarrow A \uparrow H \rightarrow H \downarrow B & | & H \longleftarrow \bullet \\
A \longleftarrow H \rightarrow A \uparrow A \leftarrow C & | & A \longleftarrow \bullet \\
B \longleftarrow C \leftarrow B \downarrow B \rightarrow H & | & B \longleftarrow \bullet \\
C \longleftarrow B \downarrow C \leftarrow C \uparrow A & | & C \longleftarrow \bullet,
\end{array}
$$

Algorithm 9.2: Scheme of an adaptive Hilbert traversal

```
Procedure H() begin
    if node cell is a leaf then
        // Execute task on current position
        execute (...);
    else
        refineH ();
        A(); up();
        H(); right();
        H(); down();
        B();
        coarsenH ();
    end
end
```

into a traversal algorithm, we required an additional production rule to prescribe a uniform level of refinement. Now, to keep track of the refinement level of a non-terminal in the generated strings, we require additional information. Such information can be obtained from two additional terminals that model refinement and coarsening – we will simply use parentheses, and obtain the following productions:

$$\begin{array}{lcl} H \longleftarrow (A \uparrow H \rightarrow H \downarrow B) & | & H \longleftarrow \bullet \\ A \longleftarrow (H \rightarrow A \uparrow A \leftarrow C) & | & A \longleftarrow \bullet \\ B \longleftarrow (C \leftarrow B \downarrow B \rightarrow H) & | & B \longleftarrow \bullet \\ C \longleftarrow (B \downarrow C \leftarrow C \uparrow A) & | & C \longleftarrow \bullet. \end{array}$$

These productions lead to a classical context-free grammar. As we can thus expand any of the non-terminals on any level, we can model arbitrarily refined quadtrees and define a Hilbert order on the respective quadtree cells. The improved traversal Algorithm 9.2 is directly obtained from the context-free grammar by matching the new terminal characters with functions `refineH()` and `coarsenH()`. During the traversal, the functions `refineH()` and `coarsenH()` have to perform two tasks:

1. Reduce (or increase) the step size for the terminal steps `up()`, `down()`, etc., before (or after) executing the calls for the child cells.
2. Move from the center of a parent cell to the center of the first child cell (`refineH`), or move from the center of the last child cell back to the center of the parent cell (`coarsenH`) – the respective operations therefore depend on the current pattern, e.g. H.

The adaptive Hilbert curve in Fig. 9.6 can be produced with Algorithm 9.2, if the nodes of the polygon are obtained from the positions successively marked by the procedure `exec()`.

Algorithm 9.3: Adaptive Hilbert traversal of a bitstream-encoded quadtree

```
Procedure H()
    Data: bitstream: bitstream representation of spacetree;
          streamptr: current position
begin
    // move to next element in bitstream
    streamptr := streamptr + 1;
    // check whether current node is leaf
    if bitstream[streamptr] then
        // execute task on current position
        execute (...);
    else
        // read and process all child nodes
        refineH ();
        A(); up();
        H(); right();
        H(); down();
        B();
        coarsenH ();
    end
end
```

9.2.3 *Refinement Information as Bitstreams*

Algorithm 9.2 performs a depth-first traversal of the quadtree, where in each level the order of the children is prescribed by the Hilbert grammar. Hence, the relative position of the child cells is given, and the only information required to restore the entire structure of the quadtree grid is whether a tree node is refined or not. We can code this information in a single bit. Moreover, the respective refinement bits will be required exactly in the order defined by the depth-first traversal. We can thus simply store the refinement bits in a *bitstream*, which in the simplest case could be implemented as an array of bits. The result is Algorithm 9.3, which can thus be used for quadtree traversals. Figure 9.7 illustrates the refinement bits for our quadtree example, and also shows a typical part of the refinement bitstream. Note that we've omitted the information whether a terminal quadtree cell is entirely inside (grey) or outside (white) the domain. Such information could be inserted after the respective 0-bit. However, we can also store a separate array that contains all this information, where the terminal cells are again stored in Hilbert order (compatible to the depth first traversal). To model more complicated, realistic objects via a quadtree, we can generalise this additional array, and use it to store additional material information on the object in each of the array elements.

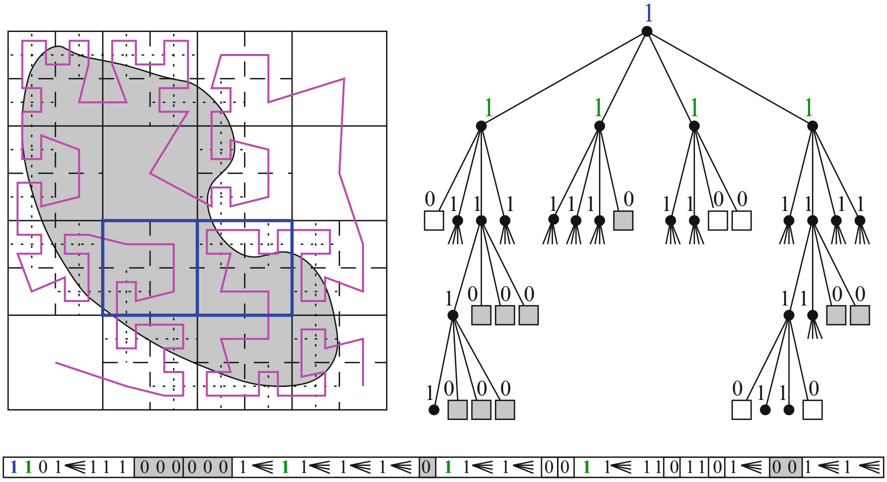


Fig. 9.7 Traversal of a quadtree grid in Hilbert order together with the refinement information (as annotation for the quadtree and as refinement bitstream)

9.3 Sequentialisation of Adaptive Grids Using Space-Filling Curves

The sequentialisation of quadtree grids according to a Hilbert order can, of course, be generalised to most of the other space-filling curves discussed in this book – provided we use adaptive grids that are compatible with their construction process.

Figure 9.8 is already known from our introductory chapter (as Fig. 1.4) – we now recognise it as a quadtree grid that is sequentialised via Morton order (or Lebesgue curve).

Figure 9.9 provides a simple example of a 3^3-spacetree grid that is sequentialised by a 3D Peano curve.

In Sects. 6.1 and 8.3, we discussed Sierpinski curves in 2D and 3D. Just as the spacetree grids, their construction results from a recursive subdivision process of the grid cells – however, instead of squares and cubes, we used triangles and tetrahedra as cells. Again, it is straightforward to introduce Sierpinski-based sequentialisations for adaptive triangular and tetrahedral grids. Figure 9.10 shows such an adaptive triangular grid, the corresponding representation via a (binary) refinement tree, and the sequentialisation of the grid cells along a Sierpinski curve.

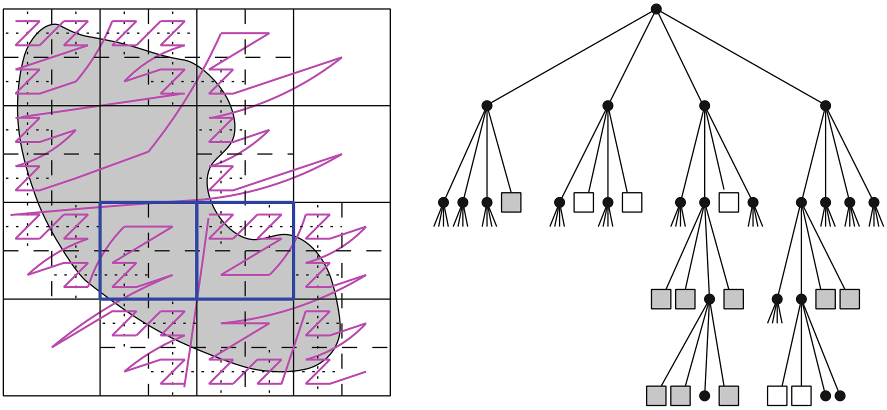


Fig. 9.8 Sequentialisation of a quadtree grid using Morton order

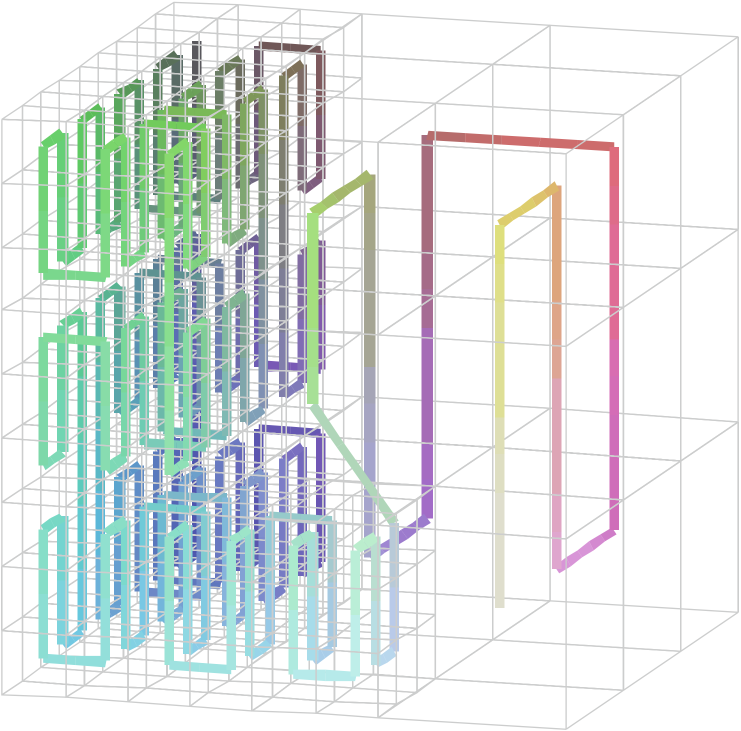


Fig. 9.9 Sequentialisation of a 3D spacetree using a Peano curve

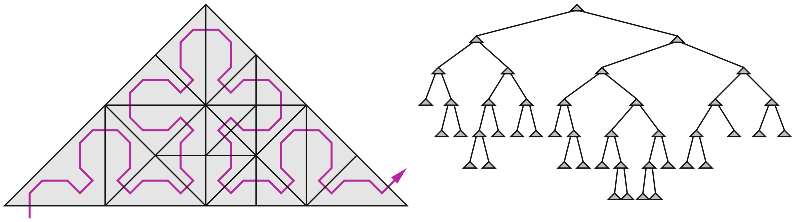


Fig. 9.10 An adaptive triangular grid, its corresponding refinement tree, and the sequentialisation of the grid cells along a Sierpinski curve

References and Further Readings

The concept of quadtrees appeared in the early 1970s in image processing – see for example Klinger and Dyer [148, 149] or Hunter and Steiglitz [136] (who gave a proof that the number of leaf cells depends on the length of the boundary of an object), but also the approaches by Warnock [263] or by Tanimoto and Pavlidis [254] (who used only fully balanced quadtrees). In the context of data bases, and roughly at the same time, quaternary trees were constructed to organise point data (tuples), such as by Finkel and Bentley [87] and Bentley and Stanat [142]. A bitstream encoding for quadtrees, such as discussed in Sect. 9.2.3, was first presented by Kawaguchi [147] (for use in image compression). See Samet [235] for a review on the history of quadtrees, and also on algorithms on quadtrees. Samet also discussed recursive refinement of triangles and other geometric objects. With the advent of 3D computer graphics, quadtrees were extended to octrees [138, 178, 179] and higher-dimensional hyper-octrees [279].

As binary or quaternary encodings of quadtrees [2], especially via depth-first sequentialisation of the leaves [96, 201], directly lead to Morton order [153], and Z order [202, 204], i.e., to the Lebesgue space-filling curve, respective connections were soon pointed out. An adaptive Hilbert indexing on a quadtree structure was probably first described by Quinqueton and Berthod [224], though they did not use the term quadtree.

Spacetrees and Refinement Trees in Scientific Computing

In computational science, the use of quadtrees and octrees grew popular with the presentation of the Barnes-Hut algorithm [28] and similar methods to efficiently compute long-range interactions of particles, especially in astrophysics. Octrees were also utilised for grid generation – both, to generate octree grids for discretisation [4, 68] but also as helper structure to efficiently generate unstructured meshes [237, 241, 243]. Aftosmis et al. [5] used Hilbert and Morton order to optimise various computational tasks (mesh coarsening, partitioning, inter-mesh interpolation) on adaptive quadtree and octree meshes for computational fluid dynamics (see also [275] for a similar approach). We will save all references on parallelisation, which is surely the best established application of space-filling curves, for the respective section in Chap. 10. In Chaps. 13 and 14, we will list approaches that improve cache performance using space-filling curves.

The presentation of *refinement trees*, as given in this section, is based on the respective partitioning method introduced by Mitchell [189], who discussed tree-based definition of space-filling orders for quadtree and octree grids, as well as for adaptive triangular and tetrahedral grids that result from bisection or quadrisection/octisection refinement. See also the reference sections of Chap. 12 (space-filling orders on tetrahedral meshes) and, in particular, of Chap. 10 (parallel partitioning using refinement trees). Maubach [177] also presented an algorithm to define a Sierpinski order on an adaptive triangular grid, which is however, more complicated than Mitchell's refinement-tree approach or an algorithm similar to 9.3.

Computer Graphics: Triangular Grids and Triangle Strips

In computer graphics, triangular meshes are especially popular, as certain basic operations are simpler to perform on triangles, and because hardware accelerators have a long tradition of speeding up the processing on triangular grids. The use of structured refinement to create right-triangular grids started out from quadtree grids. An early work by Von Herzen and Barr [261], for example, described how to obtain triangular meshes from *restricted* quadtrees, which allow at most a 2:1 size-difference between neighbouring quadtree cells (see also Exercise 9.1). Recursive bisection and respective triangle trees were, for example, introduced by Duchaineau et al. [80] (who discuss refinement cascades), Lindstrom et al. [163], or by Evans et al. [84]. Hebert et al. [126] used newest vertex bisection represented via binary trees and binary codes, and studied traversal algorithms for respective grid points. Triangle trees that are not based on bisection refinement were, for example, studied by de Floriani and Puppo [73] or Lee and Samet [159] (see also the references therein).

Sierpinski orders on recursively structured grids are especially used in the context of *triangle strips*, i.e., edge-connected sequences of triangle cells, which can be efficiently stored and processed by graphic libraries and hardware. Pajarola [207] generated such triangle strips (equivalent to Sierpinski order) on restricted quadtree triangulations; similar approaches were followed by Lindstrom and Pascucci [164], Velho and Gomes [260], or Hwa et al. [137]. Pascucci [211] also discussed the use of a 3D Sierpinski curve (as given in Sect. 12.2.2) to generate tetrahedral strips. The construction of triangle strips on unstructured and structured meshes were discussed by Bartholdi and Goldsman [30].

What's next?

- The combination of spacetrees and space-filling curves provides the basis for many applications of space-filling curves in scientific computing. The next chapter will discuss parallelisation as maybe the most important.
- Further applications will be presented in Chaps. 13 and 14. You can have a short look at the basics, now, but to discuss them in details requires an understanding of locality properties of space-filling curves – which is the topic of Chap. 11.

Exercises

9.1. Consider a *restricted* quadtree, i.e., a quadtree where neighbouring quadtree cells allow at most a size-difference of 2:1. Starting from this adaptive grid, construct a *conforming* triangular grid, i.e., where all grid points are vertices of the triangular cells (triangle vertices are not allowed to lie on an edge of the neighbour triangle). Try to develop a formal construction scheme that leads to grids that are compliant with newest vertex bisection.

9.2. The bitstream representation of a spacetree or refinement tree is motivated by a traversal of the entire tree – including all inner nodes of the tree. Discuss how the data structure has to be modified or extended to allow a traversal that only visits the leaves of the tree.

9.3. A *triangle strip* can be represented as a sequence of vertices $A, B, C, D, E, \dots,$ which models the triangle sequence $[A, B, C]$, $[B, C, D]$, $[C, D, E]$, etc. Check that, using this notation, it is not possible to represent a triangle sequence in Sierpinski order via a triangle strip. There are two approaches to "cure" this problem – for example, by inserting vertices multiple times (thus allowing additional, possibly "degenerate", triangles) or by introducing a *swap* command that exchanges two vertices in the vertex sequence. For both possibilities, show how to generate a triangle strip from Sierpinski order.

Chapter 10
Parallelisation with Space-Filling Curves

Applications for the fastest and largest supercomputers nowadays originate to a large extent from scientific computing – in particular, from numerical simulations of phenomena and processes in science and engineering. Examples can be found in many diverse areas, such as the computation of aerodynamic properties of vehicles or aircraft, the prediction of weather or climate phenomena, in molecular physics and chemistry, but also in astrophysics and geoscience.

As the accuracy of these simulations is determined by the number of unknowns used in the computational model, simulation codes often have to be able to deal with hundreds of millions or billions of unknowns. Both the computation time and the memory requirements demand the use of parallel supercomputers to make such large problems tractable. The computational scientist's job is then to parallelise the numerical simulation. The standard approach is to partition the computational domain into equal parts, and distribute these partitions onto the available computing cores. We will introduce this problem for a well-known example: the heat equation.

10.1 Parallel Computation of the Heat Distribution on a Metal Plate

Let us remember the heat distribution example we used in the introduction. For a metal plate that is discretised via a computational grid, such as illustrated in Fig. 10.1, we want to compute the approximate temperatures at the respective grid points.

Again, we can set up a system of linear equations for the temperature values. For the Cartesian grid, we obtain the system known from Eq. (1.1):

$$u_{i,j} - \frac{1}{4}\left(u_{i-1,j} + u_{i,j-1} + u_{i+1,j} + u_{i,j+1}\right) = f_{i,j} \qquad \text{for all } i, j, \qquad (10.1)$$

M. Bader, *Space-Filling Curves*, Texts in Computational Science and Engineering 9,
DOI 10.1007/978-3-642-31046-1_10, © Springer-Verlag Berlin Heidelberg 2013

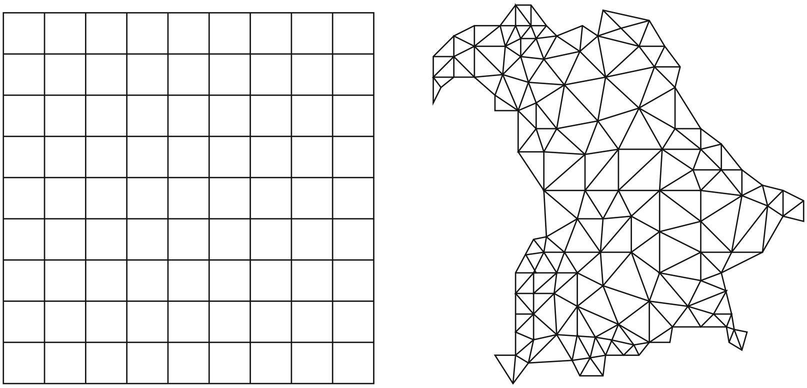

(a) Cartesian Grid (b) Triangular Grid

Fig. 10.1 Two computational grids for the metal plate problem: a simple Cartesian grid (*left image*), and an unstructured triangular grid (*right image*)

where $u_{i,j}$ is the unknown temperature at grid point (i, j), and $f_{i,j}$ models a heat source or drain at this grid point.

As we want to compute the temperature distribution as accurately as possible, we try to increase the number of unknowns as far as our computer allows. To limit the computational load for each computing core of our (super-)computer, we distribute our problem in smaller *partitions*. The partitions will be distributed to the cores (and also to the available main memory, if distributed memory is used), solved in parallel, and then combined to a global solution. Neglecting the potential numerical issues involved with such a combination of individual solutions, we also have to ask ourselves how the partitions should be chosen. In particular, what criteria should be satisfied by an efficient partitioning of the computational grid?

Criteria for an Efficient Partitioning

In parallel computing, there are two “killer criteria” that are most of all responsible for losing computational efficiency. First, the computing time lost due to *communication* between processors, which is by orders of magnitude slower than computation itself. And second, a bad *load distribution*, i.e. a non-uniform distribution of the total computational work to the available processors, which will cause processors to be idle and waiting for other processors to finish. For the creation of parallel partitions in our application, this poses two requirements of fundamental importance:

- *Each partition shall contain the same number of unknowns!*
  Usually, the computational costs are mostly determined by the number of unknowns. Hence, a uniform distribution of unknowns will usually ensure a

uniform distribution of the computational work. Note that we have to adapt this requirement as soon as the amount of work per unknown is no longer uniform. In addition, further computational resources, most of all memory, might also have an influence on the load distribution.

- *The boundaries of the partitions shall be as short as possible!*
  In our system of Eqs. (10.1), only unknowns that correspond to neighbouring grid points are coupled by the system. Hence, for all typical computations (solving the system of equations, computation of the residual), we will access those neighbouring unknowns at the same time. It is therefore important to have as many unknowns available on the local processor, i.e. in the local partition, as possible. For all unknowns outside of the local partition, costly communication would be necessary to retain a uniform value of such unknowns across all partitions. Keeping the partition boundaries short therefore *reduces communication* between processors.
  Considering the geometry of the partitions, the requirement for short boundaries will prefer compact partitions over stretched ones. Note that an important additional requirement can also be to keep the number of neighbours small, in order to reduce the number of data exchanges between pairs of processors. This is especially important, if establishing a connection between processes is expensive.

Depending on the specific application, further requirements may be important. Similarly, the relative importance of the different requirements may strongly depend on the application. We will especially consider *adaptive* grids. There, the number of grid points and unknowns is increased at critical regions, in order to achieve an improved accuracy there. Such an increased resolution might be necessary to capture small details of the computational geometry, but it can also be required for numerical reasons. For example, an *error estimator* (or error indicator) could be used to determine regions of large error in the computational domain, where the the grid should be refined and thus further unknowns invested. In time-dependent simulations, the regions of adaptive refinement might change throughout the simulation, and a *dynamically adaptive* refinement and coarsening may become necessary.

Such adaptive or dynamically adaptive grids will pose further demands on a successful partitioning algorithm:

- For adaptive grids, a partitioning algorithm must not be based solely on the geometry of the computational domain, but has to consider the grid refinement and (possibly changing) position of the unknowns.
- If the initial load distribution can be considerably changed by dynamical refinement and coarsening, a quick update of the partitions or even a fast repartitioning must be possible. Hence, in addition to computing an initial load distribution, efficient *load balancing* should be possible.
- For comparably small changes of the number of unknowns per partition, the shape of the partitions should change only slightly, as well. It is desirable to retain as many unknowns in their old partition as possible, because migrating unknowns to new partitions requires expensive communication.

10.2 Partitioning with Space-Filling Curves

Space-filling curves offer two properties that make them particularly attractive to be used for parallelisation:

- Space-filling curves provide a *sequential order* on a multidimensional computational domain and on the corresponding grid cells or particles used for simulations. To subdivide such a sequentialised list into partitions of equal size is a trivial task, and even if the partitioning criteria are more complicated, transforming the problem into 1D will considerably simplify the problem.
- Space-filling curves *preserve locality and neighbour relations.* Two points in space with an only slightly different Hilbert or Peano index will also be close neighbours in the image space. Hence, partitions that combine neighbouring points in the index space will correspond to a local point set in the image space, as well.

Hence, we can expect that a parallelisation based on space-filling curves could be able to satisfy the two main requirements for partitions – uniform load distribution and compact partitions. To actually compute the partitions, we will discuss two main options: an approach via the computation of the indices generated by the space-filling curve, but also traversal-based algorithms that build on the recursive structure of the construction. The quality of the resulting partitions will be discussed in detail in Chap. 11.

Partitioning by Using the Indices of the Grid Points

With the inverse mapping of a space-filling curve, we can compute an index for each grid point of a discretisation mesh. These real-valued indices impose a sequential order on the grid points, which we can use to define partitions for parallelisation. A straightforward option is to first sort the grid points according to the computed indices, and split this sorted array of grid points into equal-sized parts (similarly, we could sort the grid cells according to the indices of their centre points). A prototype implementation to compute partitions, for example based on the Peano curve, is then given by the following steps:

1. For each grid point g_i, compute the Peano index $\bar{p}^{-1}(g_i)$.
2. Sort the vector (array/list/...) of grid points g according to the indices $\bar{p}^{-1}(g_i)$.
3. Partition the vector g of N grid points into K partitions:

$$g^{(k)} := \left(g_{(k-1)N/K}, \dots, g_{kN/K-1}\right) \qquad \text{for } k = 1, \dots, K.$$

While step 3 will require at most $\mathcal{O}(N)$ operations (to place each grid point into its partition), standard algorithms for sorting the grid points (using Mergesort or Quicksort, e.g.) will require $\mathcal{O}(N \log N)$ operations. The computational effort for

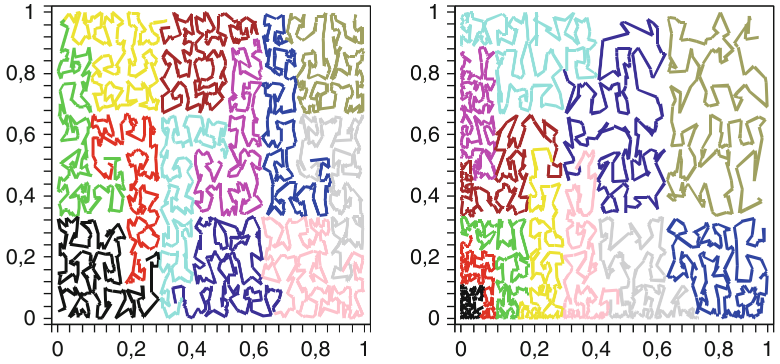


Fig. 10.2 Partitioning of a set of randomly chosen grid points according to their Peano index. In the example to the *right*, the points are concentrated towards the *lower-left* corner

the Peano indices, at first glance, seems to be linearly dependent on the number of grid points. However, remember that the computation of $\bar{p}^{-1}(g_i)$ for an individual point g_i depends on the number of digits in the resulting index. If the number of grid points N grows, we also need to increase the accuracy of the index computation, but even in the worst case, $\mathcal{O}(\log N)$ digits will be sufficient. This leads to a computational effort for $\mathcal{O}(N \log N)$ for the entire partitioning problem.

An advantage of the sorting approach is that we do not have any restrictions on the structure of the grid. Even for unstructured point sets, such as given in Fig. 10.2, a partitioning using the sorting approach is possible without changes. Note, however, that the partitioning itself typically has to be executed in parallel. Hence, we require a parallel sorting algorithm, and also the distribution of points to partitions needs to be parallelised. For choosing a suitable parallel sorting algorithm, the following properties should therefore be considered:

- Typically, only part of the computational grid will change, which means that the grid cells will be presorted to a large degree. A sorting algorithm should be efficient for such situations, and must not deteriorate to a slow algorithm in that case (which would be the case for a straightforward Quicksort algorithm).
- Due to the index computations, we are not restricted to comparison-based sorting algorithms, but may use algorithms like Bucketsort, which have an $\mathcal{O}(N)$-complexity in the average case.
- If we choose a low precision for the index computations, the sorting algorithm should be stable in the sense that cells that have the same index will not be exchanged by the sorting algorithm. In that way, such cells will not move to other partitions unnecessarily, for example during repartitioning.

For examples of suitable sorting algorithms, see the references section of this chapter. See also Exercise 10.1, which discusses comparison algorithms to efficiently determine the relative space-filling-curve order of two points.

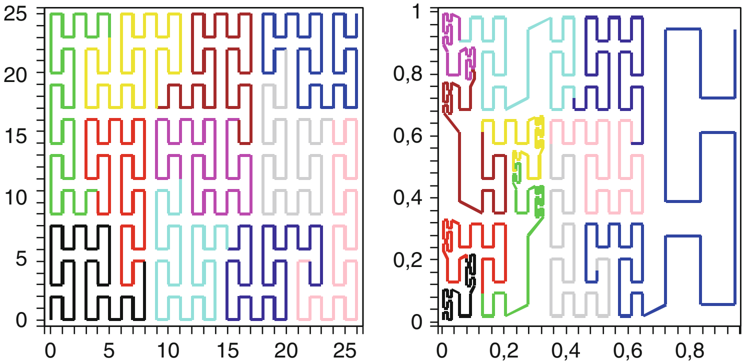


Fig. 10.3 Traversal-based partitioning of two rectangular grids using the Peano curve. In the *right* image, an example using a recursively structured grid is shown

Traversal-Based Partitioning

For a uniform grid, such as the Cartesian grid given in Fig. 10.1, but also for any grid that is described by a spacetree or refinement tree, a sorting-based partitioning algorithm is likely to be inefficient, because the special structure of the grid is not exploited. A more efficient approach can be based on using grid traversals, as they are discussed in Sects. 3.2 (for uniform grids) and 9.2 (for adaptive grids). A prototype algorithm is then given by the following steps:

1. Traverse the grid points g_i along a space-filling curve, and label the grid points by a respective incremental index n_i.
2. Use the index n_i to place each grid point g_i into a partition, such that

$$\frac{(k-1)N}{K} \leq n_i < \frac{kN}{K}.$$

If we already know the total number N of grid points, one traversal is sufficient, otherwise we will need an initial traversal to count the grid points. In practice, we will keep track of the total number of grids points during refinement and coarsening of the grid cells. We will also update the indexing during such traversals.

The computational effort of the resulting partitioning algorithm is $\mathcal{O}(N)$, as during the traversal, each grid point is processed exactly once, and both the numbering and the placement in a partition can be implemented with $\mathcal{O}(1)$ operations. The effort for the traversal itself is also $\mathcal{O}(N)$. Hence, the traversal-based partitioning is cheaper (by a logarithmic factor) than using index-based sorting. The advantage is obtained by using a structured grid; however, note that the grid may be recursively structured and thus still be a fully adaptive grid, such as illustrated in Fig. 10.3. For unstructured grids, it is possible to generate and maintain a recursively structured grid as an organisational data structure, and thus also use the traversal-based partitioning for unstructured grids or particle-based data. However, note that such an organisational structure can usually not be set up with $\mathcal{O}(N)$ effort for an unstructured mesh, so it has to be generated alongside the unstructured mesh.

10.3 Partitioning and Load-Balancing Based on Refinement Trees and Space-Filling Curves

Whenever we are faced with a dynamically adaptive computation, the number of grid cells might change strongly across the computational domain. Then, even if we start with a perfectly load-balanced grid, the partitions may become unbalanced after a certain number of refinement and coarsening steps. To re-balance the computational load, we can follow either of the following basic principles:

- We can re-compute the partitions from scratch. Such a repartitioning is worthwhile, if we expect substantial changes of the grid partitions. We then even consider that entire partitions (in contrast to just small parts of them) have to be relocated to other cores. It is only feasible, though, if the costs for the repartitioning are not too high, or at least if the repartitioning is not required too often.
- We can restrict ourselves to exchanging grid cells between neighbouring partitions. In that case, two neighbouring partitions will determine the difference of their cell numbers (or computational load, in general) and will then try to level out that difference by exchanging a suitable number of cells (half of the difference, for example). Such approaches are often referred to as *diffusion approaches* and require a couple of load balancing steps to level out large imbalances. In a strict space-filling-curve setting, we can even restrict the exchange of cells to neighbours in the sequential order given by the space-filling curve. Then, every partition will only exchange cells with its predecessor and successor in the space-filing order, which further simplifies the algorithms.

If we follow a refinement-tree approach to represent an adaptive grid, a repartitioning step can be prepared by augmenting the refinement tree by information on the computational load in each node. In Fig. 10.4, we have augmented each node of the tree by the number of leaves in the corresponding subtree. Hence, we assume that each leaf, and its corresponding grid cell, cause the same computational load. By tracking and updating such load information on each subtree, we can, during repartitioning, determine for any partition what grid cells it will keep, which cells it will forward to other process (and to which process), and whether it will obtain additional cells from other processes. For the the case given in Fig. 10.4, we can determine the partitions via the following steps:

1. The root node carries information on the total computational load (19), and thus provides us with the three partitions $\{1, \ldots, 7\}$, $\{8, \ldots, 13\}$, and $\{14, \ldots, 19\}$.

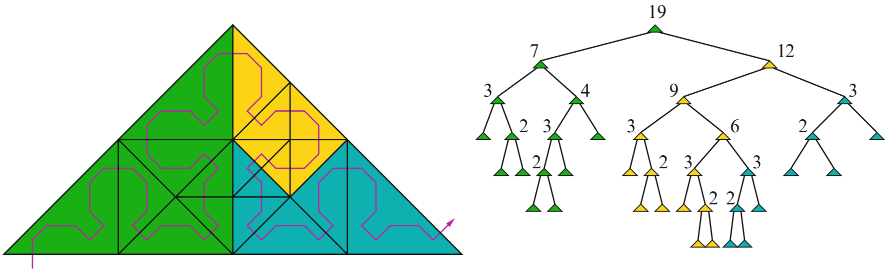


Fig. 10.4 Three partitions and their respective refinement tree – the inner nodes of the refinement tree are augmented by respective load information (leaf nodes have load 1)

2. From the load information on level two, we obtain that the left subtree contains leaf numbers $\{1, \ldots, 7\}$, while the right subtree contains leaf numbers $\{8, \ldots, 19\}$.
3. Thus, the first partition, $\{1, \ldots, 7\}$ (green), consists of the left subtree. Partitions $\{8, \ldots, 13\}$ and $\{14, \ldots, 19\}$ are in the right subtree.
4. In the node with load 12, we consider the loads of the children, 9 and 3. We conclude that the left subtree will contain the entire second partition (yellow, 6 cells) plus the first 3 cells of the third partition (cyan). And so on…

Hence, in every node of the tree, we have as information the index range of the leaf nodes that are contained in the tree, and we know the boundaries of the partitions, which we can compute from the root node. See also Exercise 10.3, which discusses how to turn this construction into a formal algorithm.

A corresponding parallel algorithm requires the load information to be replicated on processors. As a minimum, we need to store the load of the root node (i.e., the total number of leaves) on all processes. In addition, a parent node requires the load information of its children to determine the partition sizes. Figure 10.5 illustrates the information required on each partition. We have to embed a given partition into a complete refinement tree that starts from the root node, in a similar way as if we would describe a complicated domain by a refinement tree. Note that the load information in siblings is replicated in all corresponding processes. It will therefore often be more convenient to store that information in the respective parent nodes.

10.4 Subtree-Based Load Distribution

So far, our partitioning approaches have been cell-oriented: for a given cell, we determined the corresponding partition, which was given by the space-filling-curve index of the cell. As a result, we will usually obtain partitions that consist of fractions of a subtree. The respective cells are contiguous with respect to the order given by the space-filing curve; however, they may consist of many different subtrees, which might not belong to a compact, local subtree.

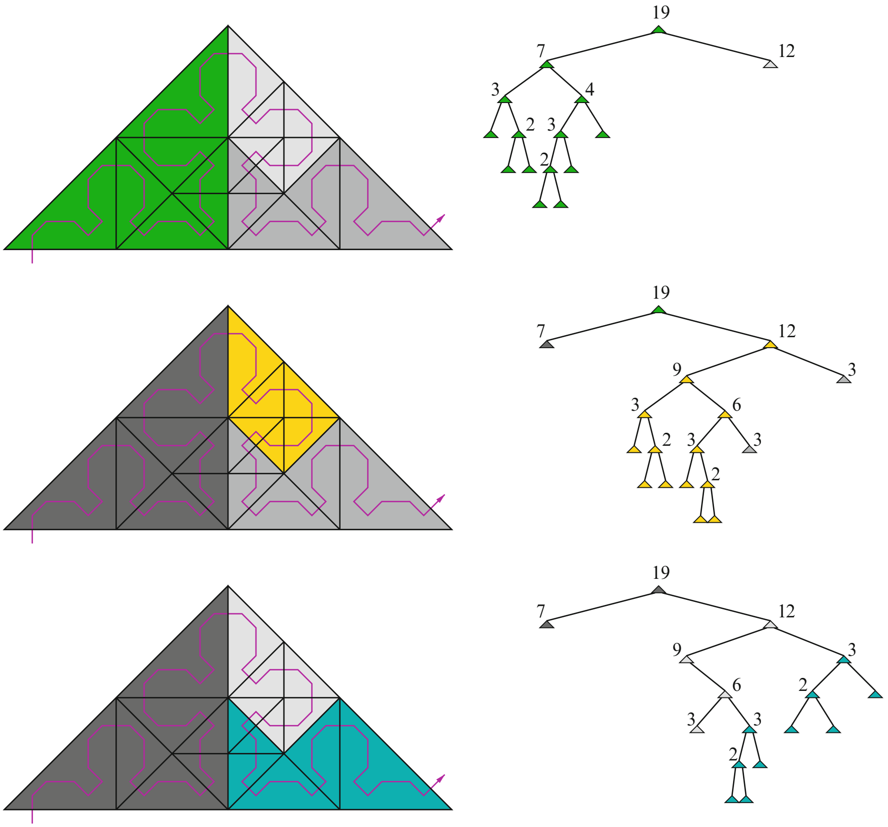

19
7
12
3
4
2
3
2
19
7
12
9
3
3
6
2
3
3
2
19
7
12
9
3
6
2
3
3
2

Fig. 10.5 The parallel partitions as in Fig. 10.4, each of which is embedded into a corresponding refinement tree with load information for partitioning

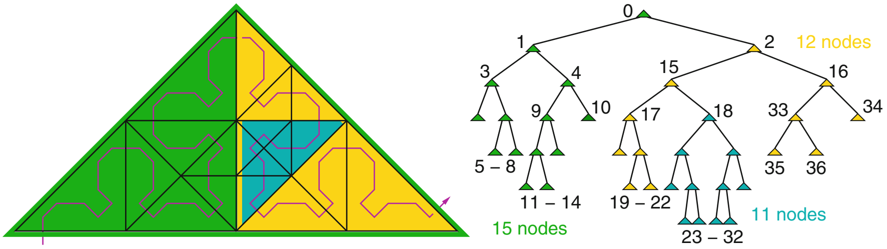

0
1
2
12 nodes
3
4
15
16
9
10
17
18
33
34
5 – 8
35
36
11 – 14
19 – 22
11 nodes
15 nodes
23 – 32

Fig. 10.6 Subtree-oriented partitioning

An alternative approach is therefore to switch to a subtree-oriented partitioning, as illustrated in Fig. 10.6. There, we strive to construct partitions that consist of entire subtrees that are as large as possible. In each inner node of the refinement tree, we may decide to split off one of the subtrees and put the resulting partition onto a different processor – similarly, we can split off several subtrees to other processors, if we consider the case of non-binary trees, such as quadtrees or octrees.

Master-Slave Structure and Quality of Partitions

The main motivation for this subtree-oriented approach is to keep the communication structures between partitions, but also the load balancing mechanism, as simple and as local as possible. If a partition becomes too big, it can decide – independent of all other processes and partitions – to split off a subtree of appropriate size. Hence, in a *master-slave* sense, the master can successively delegate work to slaves. The slaves may again become local masters and delegate subtrees to other processes.

As the sequence of cells in the subtrees is still given by the space-filling curve, the partitions will have the same locality properties as for the previous algorithms. However, as entire subtrees are split off, the shape of the boundaries is much more regular than it would be for standard space-filling-curve approaches. We can expect that the partition boundaries between a master and its slave processes consist of just a few $(d-1)$-dimensional hyperplanes, which can even lead to a reduced amount of communication (see the discussion of the edge-cut of partitions, in Sect. 11.2.1). In particular, however, we obtain a communication pattern that is easier to implement. If a subtree is split off as a new partition, the newly generated partition boundary will be between master and slave. Hence, any given section of the partition boundary can be tracked back to a former master. Moreover, every time a partition is split off, the master process can forward all required boundary information to the slave, including with which neighbour processes it will need to communicate.

Load Balancing and Work Pool Approach for Parallelisation

The orientation towards subtrees, however, also leads to certain restrictions: If we only allow one subtree per partition, we will not necessarily be able to obtain a perfectly balanced partitioning – see the example in Fig. 10.6. Even if we allow several partitions per processor, we will strive to make the subtrees as large as possible. The resulting coarse granularity of partition sizes, however, might then still lead to certain imbalances. Hence, the subtree-oriented approach will only pay off, if these load imbalances can be limited, and are countered by gains due to the simpler communication structure.

A possible solution to finding processes that can take over work from masters is a work-pool approach. The subtrees are announced to a work pool, from which “lazy” workers (slaves) may pick up additional work. If the work pool is distributed across the available compute cores (as a collection of work queues, for example), the respective approach is equivalent to classical *work stealing* approaches. See the references section of this chapter for additional information on such approaches.

10.5 Partitioning on Sequentialised Refinement Trees

In Sects. 9.2 and 9.3, we discussed how recursively structured adaptive grids can be stored via sequentialised bitstreams that encode the structure of the corresponding refinement tree of the grid. The inherently sequential processing of such a bitstream, however, requires some additional attention, if we are working in a parallel setting:

- In a shared-memory environment, we might prefer to store the sequentialised refinement tree only once, but traverse it in parallel by several processes. Hence, we need to be able to skip grid sections that are handled by other processes. Hence, we need to augment our data structure by the required information.
- In a distributed-memory setting, we will probably choose to store a separate grid on each partition. In that case, we can keep the grid traversals simple by retaining the global grid representation, but coarsening the non-local partitions as far as possible. An interesting problem is then also how to join partitions or exchange parts of the grid in a load balancing step.

We will deal with these questions in the following two subsections.

10.5.1 Modified Depth-First Traversals for Parallelisation

Assume that we can use several threads for the processing of the unknowns on an adaptive grid, but we want to store the grid information only once and share this data structure across multiple threads. Still, each of the threads will only process a limited partition of the grid, which means that it will only traverse a limited part of the corresponding refinement tree. In each node of the refinement tree, a thread will therefore first check whether this node, i.e., at least part of the subtree below, belongs to its own partition. If not, the thread will simply skip this part of the refinement tree.

Skipping subtrees is easy as long as we store the parent-child relation explicitly. However, if we want to adopt a bitstream data structure as introduced in Sect. 9.2.3, we cannot skip a subtree, because we would need to know its exact size in the bitstream representation. Instead, we need to include this size information into the bitstream. It is sufficient to replace the single refinement bit by the number of nodes in the entire subtree, which changes the bitstream into a stream of numbers. To skip a subtree, we then simply need to skip the respective number of nodes in the stream. Figure 10.7 repeats the example in Fig. 9.7, now using the subtree size

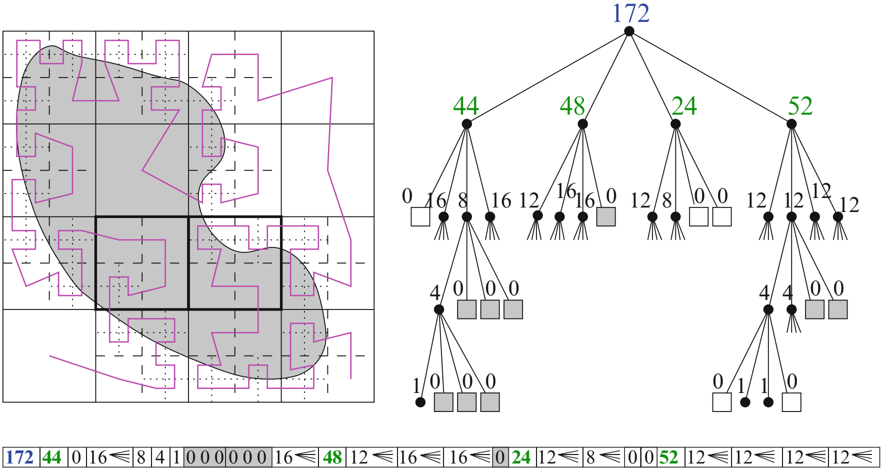


Fig. 10.7 Hilbert traversal of a quadtree grid together with the refinement stream – coding the number of nodes of each subtree as refinement information

information instead of refinement bits. Algorithm 9.3 is easily adapted to implement a traversal on such a subtree-size encoded stream – see Algorithm 10.1 in the following subsection.

Modified Depth-First Traversals

A potential disadvantage of the size-augmented storage scheme will show up especially, if we use a subtree-oriented partitioning, as described in Sect. 10.4. There, it could be advantageous to decide dynamically, in a current node, whether subtrees should be scheduled to different threads, how many subtrees should be scheduled to an available slave thread, or similar. Such scheduling decisions require the availability of all size informations for all subtrees. To retrieve the respective information, however, we need to traverse through the subtrees, or at least to perform a respective jump in the bitstream. This can be avoided, if we replace the size information of a node by the sizes of all subtrees, i.e., if we store the subtree sizes of a given node contiguously. Figure 10.8 modifies our well-known example to match this new data structure. The positions of the size informations in the data stream can be interpreted as a modified version of the depth-first traversal: for each node, we first visit all child nodes, similar to a breadth-first traversal. However, this breadth-first traversal is stopped at the child-level, and all further subtrees are visited in a depth-first fashion. The order of grid nodes induced by this modified depth-first traversal is also illustrated in Fig. 10.8. A sketch of the modified-depth-first traversal is given in Algorithm 10.1. In the second for-loop, before the if-statement,

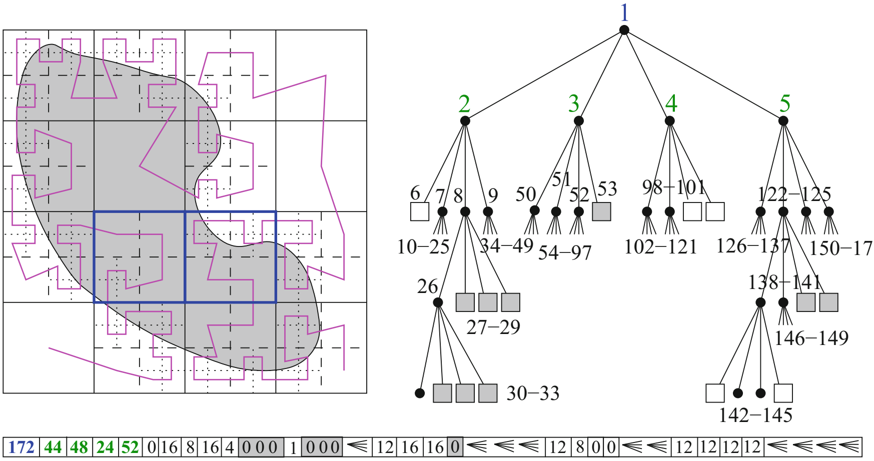


Fig. 10.8 Hilbert traversal of a quadtree grid together with the numbering of the nodes in modified depth-first search. Compare the new order within the data stream with that used in Fig. 10.7

we include a check whether the respective subtree is within a given partition. The respective decision can use the entire size information available in the array *ref*.

10.5.2 Refinement Trees for Parallel Grid Partitions

In a distributed-memory environment, we will have to store the required grid information separately for each partition. Naturally, we strive to restrict the memory requirement to a minimum, which means that we only want to store information on the part of the grid that belongs to the local partition. As already shown in Fig. 10.5, we can embed a local partition grid into the global grid by using a global grid where all non-local nodes of the refinement tree are not further expanded. Figure 10.9 updates Fig. 10.5 in the sense that we now show the corresponding grid that is stored for each partition. The respective local grids will contain all grid cells that belong to the current partition in their full resolution, but will be fully coarsened in all parts that belong to other partitions. Again, the nodes of the refinement tree can be annotated to store the number of grid cells in a respective subtree.

In a load distribution step, we will need to exchange a subgrid that contains the required number of grid cells between two grids that are represented by such local refinement trees. In a traversal-based approach, we require two steps:

- The partitions that give away grid cells need to perform a *split traversal*, where subtrees that correspond to grid parts that are no longer in the local partition will be fully coarsened. At the same time, a new local refinement tree has to be built

Algorithm 10.1: Modified depth-first traversal (in Hilbert order) of a size-encoded quadtree

```
Procedure H()
    Data: sizestream: size encoding of spacetree;
          streamptr: current position
    Variable: ref: size (sizestream elements) of the children (as array)
begin
    // read info on all childs from sizestream
    for i = 1, ..., 4 do
        streamptr := streamptr + 1;
        ref[i] := sizestream[streamptr];
    end
    refineH ();
    for i = 1, ..., 4 do
        if partition i executed by other thread then
            // skip partition in bitstream
            streamptr := streamptr + ref[i];
            continue;
        end
        if ref[i] = 0 then
            // execute task on current cell
            execute (...);
        else
            // call to recursive pattern (A, H, H, and B, respectively)
            childH(H,i);
            // step to next cell (up, right, left, and none)
            step(H,i);
        end
    end
    coarsenH ();
end
```

that embeds the grid cells that are to be transferred to the other partition (see Exercise 10.4).

- The receiving partitions now need to join the two refinement trees in a respective tree traversal. A simplified implementation of such a *join traversal* is given in Algorithm 10.2.

The split and join traversal can be generalised for splitting large partitions into two parts or joining two small partitions into a larger one, and will thus enable load balancing for scenarios where the number of available processes might change. For example, in the set-up phase of a large-scale simulation, an initial, relatively coarse grid might be successively refined to obtain the final resolution. Adopting split traversals will step-by-step increase the number of processes during such a set-up phase.

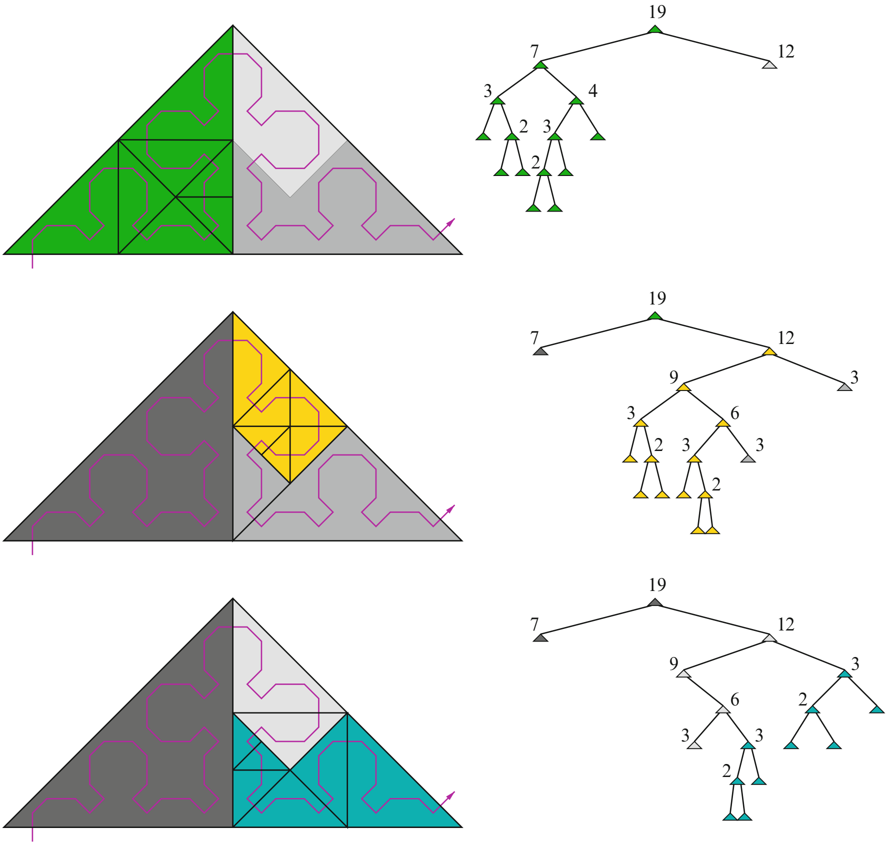


Fig. 10.9 Parallel grid partitions as in Fig. 10.5: now, each local grid contains the local grid cells in full resolution whereas the grid is fully coarsened in all parts that belong to other partitions

10.6 Data Exchange Between Partitions Defined via Space-Filling Curves

So far, we have discussed the definition of partitions via space-filling curves, and also how the respective partitions can be traversed in parallel, including necessary changes of the data structures. In this section, we will add another essential aspect of parallel processing: we will discuss how data can be exchanged between partitions. For many applications, such communication is restricted to the exchange of data at the boundary of two partitions, for example:

- Exchanging the refinement status of edges along the partition boundaries, in order to avoid hanging nodes across partition boundaries (during adaptive refinement or coarsening of the grid);

Algorithm 10.2: Traversal to join two binary refinement trees

```
Procedure join_reftree (joinStream, aStream, bStream)
    Parameter: aStream,bStream: input trees in bitstream encoding
    Result: joinStream: joined tree (in bitstream encoding, as output parameter)
    Variable: aRef,bRef: refinement status of the two current nodes
begin
    // read refinement info in bitstreams
    aRef = read (aStream) ; bRef = read (bStream) ;
    if aRef= bRef then
        // copy identical parts of the refinement tree
        append (joinStream, aRef);
        if aRef= 1 then
            join_reftree (joinStream, aStream, bStream) ;
            join_reftree (joinStream, aStream, bStream) ;
        end
    else
        // only one of the subtrees is non-empty
        if aRef= 1 then
            append (joinStream, aRef) ;
            add_reftree (joinStream, aStream) ;
            add_reftree (joinStream, aStream) ;
        else
            // bRef = 1
            append (joinStream, bRef) ;
            add_reftree (joinStream, bStream) ;
            add_reftree (joinStream, bStream) ;
        end
        // no action if aRef = bRef = 0
    end
end

Procedure add_reftree (joinStream, stream)
    Parameter: joinStream, stream: refinement trees in bitstream encoding
    Result: refinement tree stream is added to joinStream as subtree
    Variable: ref: current refinement status (of the subtree)
begin
    ref = read (stream) ;
    append (joinStream, ref) ;
    if ref= 1 then
        add_reftree (joinStream, stream) ;
        add_reftree (joinStream, stream) ;
    end
end
```

- Synchronising the unknowns on the partition boundaries as part of a domain decomposition approach with non-overlapping domains;
- Exchanging the values of so-called *ghost cells*, i.e., cells of a neighbouring partition that are replicated in the local partition.

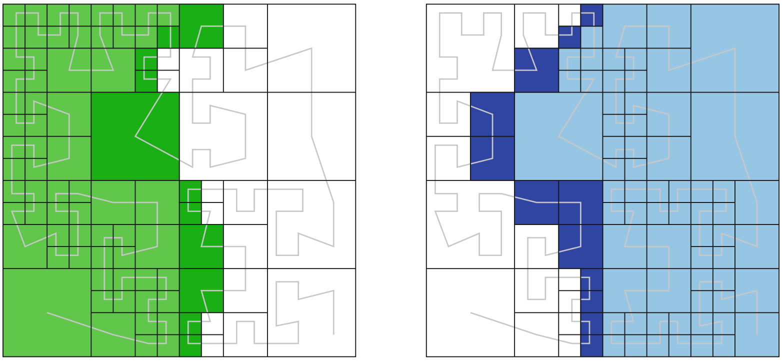


Fig. 10.10 Parallel quadtree grid partitions that share a layer of ghost cells (highlighted by using a *darker* colour) – here, ghost cells are required to share at least a common edge with any cell of the partition

The communication problem thus depends on whether two adjacent partitions will only share data lying directly on their respective partition boundaries, or whether the partitions overlap, and thus share data that is local to grid common cells.

10.6.1 Refinement-Tree Partitions Using Ghost Cells

During parallel processing, algorithms will typically access data in neighbouring grid cells to update unknowns or grid data – our system of linear Eqs. (10.1) for the heat equation being one example. To avoid communication operations each time we enter a cell close to a partition boundary, a standard approach is to replicate grid cells in a partition, if their data is required for computations by the local partition, even if these cells are treated by a neighbouring partition. Such cells are often called *ghost cells* (or halo cells). Figure 10.10 gives an example of a quadtree grid, where two parallel partitions share a layer of ghost cells. Note that each quadtree grid is embedded into the same unit domain and all cells outside the local partition and ghost cell layer are fully coarsened.

A typical numerical algorithm will perform the following steps for each partition:

1. Update the unknowns in each cell based on the unknowns in the neighbouring cells, including unknowns in ghost cells.
2. Update the values in the ghost cells, i.e., each partition will send the values of the updated unknowns to the respective neighbour partition.
3. With the updated ghost-cell unknowns, each process can proceed with step 1.

In a concrete implementation, such a ghost-cell approach has the advantage that the update of unknowns works exactly in the same way for all cells – regardless of whether they are adjacent to a partition boundary or in the interior of a partition. The parallelisation problem is thus reduced to the problem of updating the ghost layer.

Now assume that we use refinement-tree grids and traverse them according to a fixed order, given by a space-filling curve, e.g. If we use the same ordering scheme in all parallel partitions, any two partitions will traverse their common ghost cells in exactly the same order – compare the Hilbert order induced on the ghost cells in Fig. 10.10. Note that this property extends to unknowns on edges or vertices of the grid cells, as well, as long as we define a unique order in all partitions – for example, the order defined by the last access to an unknown during a traversal. The data exchange during communication between two partitions is then simplified in the sense that both partitions use the same relative order of the unknowns. In particular, it is sufficient to exchange the values of the data only, without additional information on their location.

10.6.2 Non-Overlapping Refinement-Tree Partitions

In domain composition approaches with non-overlapping domains, unknowns in parallel partitions will at most access unknowns that are located on partition boundaries, which in such contexts are therefore often referred to as a *separator*. The values of the separator unknowns then need to be approximately computed in advance – such as in so-called Schur complement methods (see [247] for an introduction to such domain decomposition methods). For certain numerical schemes (explicit time-stepping methods, for example), it is also sufficient to access separator values from the previous iterative step (or time step). This situation will often occur, if an element-wise processing of the unknowns is applied – see, for example, the case study in Chap. 14. We then have the situation that no ghost cells are required, and partitions will be directly adjacent to each other, and only share the separator, i.e., the partition boundary, with their respective neighbours. In the 2D case, the connectedness of space-filling curves will then induce a couple of helpful properties on the partitions and on the communication patterns. Thus, let's consider the following partitioning scenario:

- The computational domain is embedded into the unit square (or any other domain that is filled by a space-filling curve), and we apply a structured, but possibly adaptive grid refinement that is compatible to the construction of the space-filling curve. Grid cells that are outside of the computational domain are considered as obstacle cells, which means that they are treated as part of the grid (and of respective partitions), but either no or minimal processing is done on such cells.
- The space-filling curve compatible with this grid is assumed to be *connected* (i.e., edge-connected in 2D) and induces a respective indexing on the grid cells. Partitions are defined via index intervals (each partition consisting of exactly one index interval).

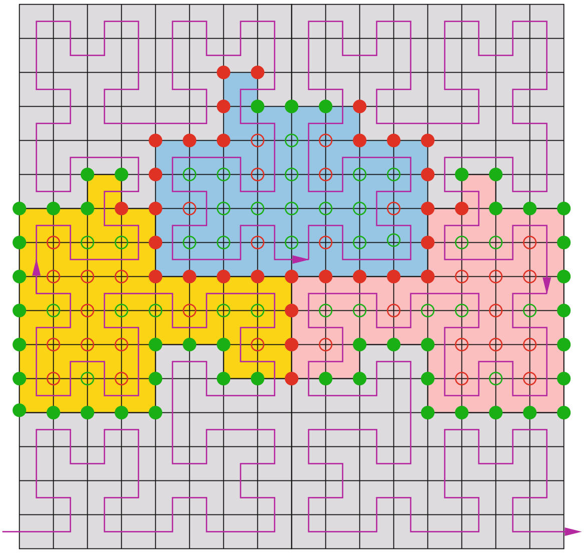


Fig. 10.11 Partitions induced by a 2D Hilbert space-filling curve – note that the node-located unknowns are split into a *left* (*green discs*) and a *right* (*red discs*) half

As we are primarily concerned with the synchronisation of the unknowns on the separators, we will, in the following, assume that unknowns are located on cell vertices. However, the discussion will also apply, if unknowns are located on cell edges. Unknowns located in the interior of cells, finally, do not pose any problems, as they are local to one partition and will not be communicated.

An important observation for 2D connected space-filling curves is then, that they will divide the node- and edge-located unknowns of the sequentialised grid into two parts – a *left* half and a *right* half, as illustrated in Fig. 10.11. In the same way, any partition – and thus any partition boundary of a given partition – will be split into a *left* and a *right* part. For parallelisation and, in particular, for the communication patterns during parallelisation, this has a couple of helpful implications:

1. As the iterations of a connected curve cannot intersect themselves, we obtain exactly one left and one right part of the boundary in each partition. Moreover, the indices of the cells adjacent to the partition boundary will grow or decrease monotonously along a partition–partition boundary. Note that the adjacent partitions will be numbered in opposite order along their common separator. This so-called *stack property* will be exploited for a much more general processing approach in Chap. 14. For parallelisation, we can utilize it for writing and reading boundary values to communication buffers. Unknowns are sent in the sequential order obtained from the traversal, and are received in inverse order by the adjacent partition.
2. Due to the monotonicity of indices, the boundary between two partitions will be contiguous. It cannot be intersected by a boundary with other partitions, because,

then, we would have a contradiction between partitions being defined via contiguous intervals, and unknowns at partition boundaries being in monotonous order.

3. As a corollary, the indices of the partitions (i.e., the numbers of the intervals that define them) are either growing or decreasing monotonously along partition boundaries, but may have occasional jumps in that sequence (consider the red boundary of the blue partition in Fig. 10.11). Still, this property can be exploited to structure the communication pattern for data exchange between all partitions, and ensure that partitions send and receive packages from neighbours in optimised order.

The left and the right part of each partition meet at the entry and exit points of the curve in this partition. We will take up this discussion in Chap. 14, where we will examine an approach that exploits the stack property at partition boundaries for additional computational tasks. There, we will also discuss the 3D case, as this, of course, is not just a natural extension of 2D.

References and Further Readings

The first applications that used space-filling curves for parallelisation again stem from N-body simulation problems [234, 245, 264], which mainly used Morton order on quadtrees and octrees – compare the references section on spacetrees. For PDE-based simulation on spacetree grids, space-filling-curve techniques for parallelisation and partitioning were quickly adopted. For example, Patra and Oden [212] used index-based graph partitioning for finite element simulations; Pilkington and Baden [217, 218] introduced an approach based on inverse SFC mappings followed by recursive bisection in 1D. Sorting-based approaches were presented by Kaddoura et al. [143] (Morton/Hilbert order on graphs that represent discretisation grids), Ou et al. [206], or Aluru and Sevilgen [10] (Z-curve). Luitjens et al. [172] especially discussed parallel sorting to generate parallel Hilbert indexing. Parallelisation approaches that use space-filling indices for hash functions were followed by Parashar and Browne [210], as well as by Griebel and Zumbusch [108, 109]. Variants of partitioning methods based on inverse mappings are discussed in [249]. To determine costs of subtrees via respective octree traversals was introduced by Flaherty et al. [88]. Scalability to very large numbers of compute cores was demonstrated in [225]. The refinement-tree approaches outlined in this section are strongly based on the work by Mitchell [189]. The subtree-oriented techniques discussed in Sect. 10.4, as well as the use of modified depth-first traversals, are based on the work of Weinzierl [265].

Interestingly, Sierpinski curves are less commonly used for parallelisation (despite their somewhat superior locality properties – see Sect. 11), even in the case of triangular grids. Respective parallelisation approaches were for example, introduced by Behrens and Zimmermann [40] (generalised Sierpinski curves for oceanic applications) and by Zumbusch [286], who also provide excellent reviews on this issue in their respective textbooks [37, 287]. Nevertheless, Hilbert or Morton order are often used even for structured triangular grids ([228], e.g.).

Locality properties of space-filling curves, and thus the quality of resulting partitions will be discussed in the next chapter – hence, see the respective references section for literature that specialised on respective theoretical aspects. Comparisons of space-filling-curve approaches with other partitioning methods, especially graph-based approaches, were presented, for example, by Hendrickson et al. [128, 129], Schloegel et al. [240], or Schamberger and Wierum [239].

Applications and Software

Parallelisation and partitioning based on space-filling curves, by now, has become state-of-the-art, and there is a multitude of various applications, and also a good number of simulation packages that exploit space-filling curves. For example, packages that use Hilbert or Morton order on octree-structured grids include PARAMESH [79], Racoon [79], Dendro [253], or p4est [55]. Zoltan [61, 76] is a load-balancing package that offers traversal-based partitioning using Morton order, Gray codes, or Hilbert curves. The Peano framework [52] (see also Chap. 14) is based on the Peano curve and respective spacetree refinement. For triangular grids, amatos [39] is a Finite Element package that uses bisection refinement and generalised Sierpinski curves for parallelisation.

As already indicated above, the parallelisation of tree algorithms for particle-type simulations was one of the earliest applications of space-filling curves for parallelisation, and is still in use for applications in astrophysics [167], for Coulomb solvers [98], or for general elliptic problems [280]. Similarly, space-filling curves have been used in molecular dynamics [14, 46, 62], for cosmological simulations using Smoothed Particle Hydrodynamics [248], or for meshfree PDE solvers [107]. Within grid-based simulations, space-filling curves were adopted as a tool for parallelisation of general PDE solvers [209] and for Finite Element problems [213, 252], including multigrid [108] or domain decomposition methods [33], or divide and conquer algorithms [12]. Specific applications include computational fluid dynamics [5, 8, 49, 52, 53], including biofluidics [225] and magnetohydrodynamics [284], and solvers for supernova simulation [60]. In the geosciences, space-filling curves have been used in the simulation of mantle convection [54], seismic wave propagation [258], as well as in atmospheric [36, 74, 195] and ocean modelling [75]. Further applications in scientific computing include matrix operations in linear algebra [19, 152], or even neural networks [63]. In classical computer science, space-filling-curves were, for example, used as an indexing scheme for processor meshes to parallelise tree searches [144], sorting algorithms [236], or problems in computational geometry [186].

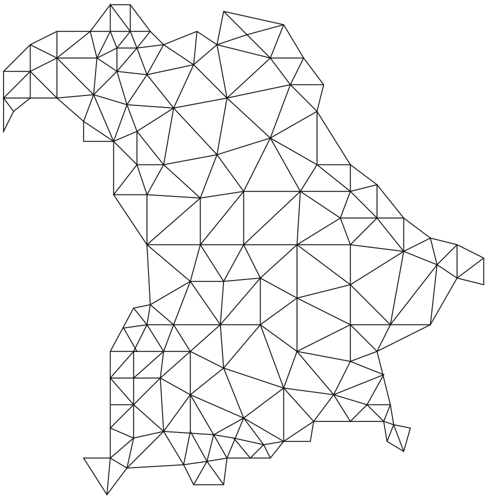


Fig. 10.12 Unstructured triangular grid covering Bavaria – can you generate a good partitioning using one of the space-filling-curve approaches?

What's next?

- The next chapter will discuss the quality of partitions defined via space-filling curves, which results from certain locality properties.
- You should not skip the locality properties entirely, but at least browse through the most important concepts (Hölder continuity, in particular), before moving on.

Exercises

10.1. Extend the algorithms derived in Exercise 4.7, such that they can be efficiently used for index comparisons of points, i.e., to determine which of two points is visited earlier by the Hilbert curve.

10.2. Take the unstructured triangular grid given in Fig. 10.12 and try to generate a numbering of the elements. You can either try a map-based approach (draw a space-filling curve on top on the grid), or impose some spacetree structure on the grid. Check the quality of your numbering by examining the generated partitions.

10.3. Examine the example for determining the parallel partitions for Fig. 10.4, and specify a prototypical algorithm to compute such partitions. Does your algorithm also work in a parallel setting, such as in Fig. 10.5, where only the partition-local part of the refinement tree is stored on a given parallel process.

10.4. Sketch an algorithm that splits a refinement tree into two partitions, reading the respective bitstream encoding, and writing the bitstream encoding of the two partitions as output.

10.5. In Sect. 10.6.2, we discussed that a space-filling curve iteration divides the node-located unknowns into two parts that are left and right of the iteration, respectively. How can you determine in each grid cell, which unknowns are left or right of the curve?

Chapter 11
Locality Properties of Space-Filling Curves

11.1 Hölder Continuity of Space-Filling Curves

The overall properties of space-filling curves, together with the results for the examples shown in Figs. 10.2 and 10.3, indicate that space-filling curves are a good heuristics for an efficient load distribution of data into compact partitions. It would, of course, be interesting to be able to quantify this property, or give a mathematical property to characterise it.

In the following section, we will see that the so-called *Hölder continuity* of space-filling curves is an appropriate tool to quantify locality properties of space-filling curves.

Definition 11.1 (Hölder continuous). A mapping $f\colon \mathcal{I} \to \mathbb{R}^n$ is called *Hölder continuous* with exponent r on the interval $\mathcal{I}$, if there is a constant $C > 0$ for all parameters $x, y \in \mathcal{I}$, such that:

$$\|f(x) - f(y)\|_2 \leq C\,|x-y|^r\,.$$

Of course, we could exchange the Euclidian norm $\|\cdot\|_2$ by another, equivalent norm in this definition.

For space-filling curves, the *Hölder continuity* provides a relation between the distance of the indices (parameters) and the distance of the image points. The distance of indices is given by the parameter distance $|x-y|$; the distance of the image points is given by $\|f(x) - f(y)\|_2$. In the following section, we will show that the Hilbert curve is Hölder continuous with exponent $r = d^{-1}$, where d denotes the dimension of the curve:

$$\|f(x) - f(y)\|_2 \leq C\,|x-y|^{1/d} = C\,\sqrt[d]{|x-y|}$$

M. Bader, *Space-Filling Curves*, Texts in Computational Science and Engineering 9,
DOI 10.1007/978-3-642-31046-1_11, © Springer-Verlag Berlin Heidelberg 2013

11.1.1 Hölder Continuity of the 3D Hilbert Curve

To prove Hölder continuity of the 3D Hilbert curve works in almost the same way as the proof for regular continuity. However, a few technical details need to be added:

1. Consider two arbitrary parameters $x, y \in \mathcal{I}$. We can then choose an n, such that $8^{-(n+1)} \leq |x - y| < 8^{-n}$.
2. 8^{-n} is exactly the length of the intervals in the n-th iteration of the 3D Hilbert construction. The interval $[x, y]$ will therefore overlap at most two intervals of the n-th level, which have to be adjacent!
3. Due to the construction of the 3D Hilbert curve, the two intervals will be mapped to two adjacent cubes with side length 2^{-n}. The function values $h(x)$ and $h(y)$ there lie within the cuboid made up from these two cubes.
4. The maximal distance between $h(x)$ and $h(y)$ is therefore limited by the space diagonal of this cuboid. The length of this diagonal is exactly

$$2^{-n} \cdot \sqrt{1^2 + 1^2 + 2^2} = 2^{-n} \cdot \sqrt{6}.$$

5. Hence, for the distance $\| f(x) - f(y)\|_2$, we obtain the following inequality:

$$\begin{aligned} \|h(x) - h(y)\|_2 &\leq 2^{-n}\sqrt{6} = (8^{-n})^{1/3}\sqrt{6} = \left(8^{-(n+1)}\right)^{1/3} 8^{1/3}\sqrt{6} \\ &\leq 2\sqrt{6}\,|x - y|^{1/3} \end{aligned}$$

   Choosing $C := 2\sqrt{6}$ and $d = 3$, we have thus proven the Hölder continuity of the 3D Hilbert curve.

From the proof, it becomes obvious that connectedness and recursivity (in particular, congruent subdomains) are sufficient properties to prove Hölder continuity. Hence, all connected, recursive space-filling curves will be Hölder continuous with exponent d^{-1}.

11.1.2 Hölder Continuity and Parallelisation

Via the inequality

$$\| f(x) - f(y)\|_2 \leq C \sqrt[d]{|x - y|},$$

the Hölder continuity of a curve provides a relation between $|x - y|$, which is the distance of the indices defined by the space-filling curve, and the distance

$\|f(x) - f(y)\|$ of the corresponding curve points. Primarily, $|x - y|$ is a distance in the parameter interval. However, we can also relate it to the corresponding section of the curve, i.e. the image of the interval $[x, y]$ under the mapping h. With this additional relation, the Hölder continuity would lead to a relation between the area of a partition, proportional to $|x - y|$, and the extent of a partition, which depends on the maximal distance $\|f(x) - f(y)\|$. Hence, we need to take a closer look at the relation between an interval $[x, y]$, and the volume of the corresponding section of the space-filling curve.

Definition 11.2. Consider the parameter representation $f : \mathcal{I} \to \mathbb{R}^n$ of a space-filling curve. We will call f *parameterised by volume (area)*, if for each interval $[x, y]$, the length $|x - y|$ of the interval is equal to the volume (area) of the image

$$f[x, y] := \{f(\xi) : \xi \in [x, y]\}.$$

We can, quite easily, prove that our previously defined mapping h for the 3D Hilbert curve is parameterised by volume:

1. For any given parameter interval $[x, y]$, we can find two sequences of intervals

   $$[a_n, b_n] \qquad \text{and} \qquad [c_n, d_n],$$

   where

   - The interval boundaries a_n, b_n, c_n, and d_n are multiples of 8^{-n}, i.e. are interval boundaries of the intervals used for the construction of the 3D Hilbert curve.
   - Each interval $[a_n, b_n]$ is the smallest interval that still contains $[x, y]$;
   - Each interval $[c_n, d_n]$ is the largest interval that is still contained in $[x, y]$.

   Note that, as a consequence, the interval boundaries will converge to x, y, i.e. $a_n, c_n \to x$ and $b_n, d_n \to y$ for $n \to \infty$. The intervals $[a_n, b_n]$ will thus serve as upper bounds, and the intervals $[c_n, d_n]$ as lower bounds in our proof.
2. Due to construction, the length of the intervals $[a_n, b_n]$ will be equal to $\alpha_n \cdot 8^{-n}$ for certain α_n. Similarly, the length of the intervals $[c_n, d_n]$ will be equal to $\gamma_n \cdot 8^{-n}$. In addition, the following inequality holds:

   $$\gamma_n \cdot 8^{-n} \le |y - x| \le \alpha_n \cdot 8^{-n}.$$

3. As the intervals $[a_n, b_n]$ and $[c_n, d_n]$ consist of the intervals used to construct the Hilbert curve, their images $h[a_n, b_n]$ and $h[c_n, d_n]$ will consist of the respective cubes of side length 2^{-n}. As the volume of each cube is given by $(2^{-n})^3 = 8^{-n}$, the volume covered by all cubes is

   $$V(f[a_n, b_n]) = \alpha_n \cdot 8^{-n} \qquad \text{and} \qquad V(f[c_n, d_n]) = \gamma_n \cdot 8^{-n}.$$

4. Due to our construction, and due to the construction of the Hilbert curve, we know that

$$f[c_n, d_n] \subset f[x, y] \subset f[a_n, b_n].$$

As a result, we obtain for the respective volumes:

$$V(f[a_n, b_n]) = \alpha_n \cdot 8^{-n} \leq V(f[x, y]) \leq V(f[c_n, d_n]) = \gamma_n \cdot 8^{-n}.$$

5. As both $\alpha_n \cdot 8^{-n}$ and $\gamma_n \cdot 8^{-n}$ converge to the length of the interval $[x, y]$, we finally obtain:

$$V(f[x, y]) = |x - y|.$$

Thus, our standard definition of the Hilbert mapping h leads to a parameter representation of the 3D Hilbert curve that is parameterised by volume.

We can again ask, whether our proof can be transferred to a larger class of space-filling curves. One of the central assumptions required for the proof is that we follow a *recursive* construction of the space-filling curve; in particular, all child domains were assumed to have an identical volume (area) and all subintervals were assumed to be of the same size. In addition, we assumed that we use intervals as parameter set – which actually prevents the Lebesgue curve (with the Cantor Set as parameter space) from being parameterised by volume. In contrast, connectedness is not required in the proof, such that the Z-curve (or a similar curve) is parameterised by volume. As recursive curves, all Peano curves and the 2D Sierpinski curve are also parameterised by volume (area). Note that the 3D Sierpinski curve introduced in Sect. 8.3 is also parameterised by volume, even though it is not strictly recursive (the shapes of the tetrahedra are not congruent to their respective parents, but have at least the same volume).

For all curves that are recursive, connected, and parameterised by volume, the length of the interval $|x - y|$ is equal to the volume of the image $f[x, y]$, such that the inequality that defines Hölder continuity,

$$\|f(x) - f(y)\|_2 \leq C \sqrt[d]{|x - y|},$$

directly relates the distance of two image points to the volume covered by the space-filling curve between these two points. The Hölder continuity therefore determines the relation between volume and extent (given by the maximum diameter, e.g.) of a partition defined by this curve. In that sense, the Hölder continuity – and in particular the involved constant C, as we will see – acts as a measure of *compactness* of partitions.

11.1.3 Discrete Locality Measures for Iterations of Space-Filling Curves

The Hölder continuity of space-filling curves also shows up when their iterations or, in general, space-filling orders induced on discrete meshes are examined. For a d-dimensional array $\{1,\dots,n\}^d$ (or the corresponding Cartesian grid), for example, such a space-filling order would define a mapping $c\colon \{1,\dots,n^d\} \to \{1,\dots,n\}^d$. A locality measure equivalent to Hölder continuity for such a discrete order is then

$$\max \left\{ \frac{\|c(i)-c(j)\|_\alpha}{\sqrt[d]{|i-j|}} : i,j \in \{1,\dots,n\}^d, i \neq j \right\}, \tag{11.1}$$

where $\|c(i)-c(j)\|_\alpha$ denotes a suitable metric, for example the Euclidian distance ($\alpha = 2$). In literature, also the following ratio is used:

$$\max \left\{ \frac{\|c(i)-c(j)\|_\alpha^d}{|i-j|} : i,j \in \{1,\dots,n\}^d, i \neq j \right\}, \tag{11.2}$$

If we require a general characterisation of the corresponding space-filling curve, we can derive discrete locality measures by taking the supremum over all iterations of the curve:

$$C_\alpha := \lim_{n\to\infty} \sup \left\{ \frac{\|c(i)-c(j)\|_\alpha}{\sqrt[d]{|i-j|}} : i,j \in \{1,\dots,n\}^d, i \neq j \right\}, \tag{11.3}$$

or, similarly:

$$\tilde{C}_\alpha := \lim_{n\to\infty} \sup \left\{ \frac{\|c(i)-c(j)\|_\alpha^d}{|i-j|} : i,j \in \{1,\dots,n\}^d, i \neq j \right\}. \tag{11.4}$$

Table 11.1 lists the locality coefficients C_α and $\tilde{C}_\alpha$ for different space-filling orders and for the following distances defined on two image points $(x_1,\dots,x_d)$ and $(y_1,\dots,y_d)$:

- The Euclidian distance: $\|x-y\|_2 := \sum (x_i - y_i)^2$
- The "Manhattan" distance: $\|x-y\|_1 := \sum |x_i - y_i|$
- The maximum distance: $\|x-y\|_\infty := \max\{|x_i - y_i|\}$

The coefficients $\tilde{C}_\alpha$ for the 2D curves are cited from the work of Haverkort and van Walderveen [123], the 3D coefficients were given by Niedermeier et al. [196] – further references are given in the last column of the table (see the references section for a more detailed discussion of these works). Note that H-index is equivalent to the Sierpinski curve.

Table 11.1 Locality measures for different space-filling curves and different distances. Exact numbers are given for analytically proven bounds; numbers are rounded to two or three significant digits, if they were obtained by experiment. Number in italics are just re-computed from the respective other coefficient, and are given for easier comparison of the numbers

	C_∞	C_2	C_1	$\tilde{C}_\infty$	$\tilde{C}_2$	$\tilde{C}_1$	Reference
2D curves:							
Hilbert	$\sqrt{6}$	$\sqrt{6}$	3	6	6	9	[34, 67, 196]
$\beta\Omega$	*2.236*	*2.236*	*3.000*	5.000	5.000	9.000	[123]
H-index	2	2	$2\sqrt{2}$	4	4	8	[196]
Morton order	∞	∞	∞	∞	∞	∞	–
Peano (Fig. 2.6)	$2\sqrt{2}$	$2\sqrt{2}$	$\sqrt{10\frac{2}{3}}$	8	8	$10\frac{2}{3}$	
Meurthe (Fig. 2.8a)	*2.309*	*2.380*	*3.266*	5.333	5.667	10.667	[123]
Sierpinski	2	2	$2\sqrt{2}$	4	4	8	[196]
Gosper flowsnake	*2.52*	*2.52*	*3.56*	6.35	6.35	12.70	[123]
3D curves:							
Hilbert a (Fig. 8.2, right)	3.08	3.22	4.62	29.2	33.2	98.4	[196]
Hilbert b (Fig. 8.2, left)	3.11	3.25	4.83	30.0	34.2	112.1	[196]
Hilbert c (Fig. 8.4)	3.11	3.25	4.83	30.0	34.2	112.1	[196]

11.2 Graph-Related Locality Measures

Any discretisation grid can also be described by a graph $G = (V, E)$, where the set of graph vertices V and respective connecting edges E are given by the vertices and edges of the grid cells. In addition, we can define a graph $\widehat{G} = (C, \widehat{E})$, where the grid cells C form the vertices of the graph, and where we define an edge in $\widehat{E}$ for any pair of cells that are direct neighbours in the grid. Such a graph is often referred to as the *dual graph* of G. Several commonly-used properties of graphs then turn into valuable properties of the represented discretisation grid:

- A *graph partitioning* will partition the vertices of the graph into disjoint sets V_i, with $V = \bigcup V_i$. The minimum and maximum sizes $|V_i|$ of the sets reflect the quality of the load distribution. If we are not dealing with element-located instead of vertex-located unknowns, such a partitioning can, of course, be defined for the dual graph $\widehat{G}$, as well.
- The *edge cut* of a graph partitioning is defined as the set of edges that need to be removed to disconnect the graph into the respective non-connected partitions. As the vertices correspond to unknowns, the edges reflect which neighbouring unknowns are required to update unknowns in solvers. Hence, the size of the edge cut is a measure for the amount of communication between partitions. Depending on the location of the unknowns, the edge cut of G or that of the dual graph $\widehat{G}$ might be the better measure.
- A graph or a graph partition is called *connected*, if it contains a path of edges between any two vertices of the graph (partition). In the same sense, we can call

a grid (or a partition) connected, if the corresponding graph is connected. Note that a face-connected space-filling curve on a grid is only possible, if the dual graph of the grid is connected. If G is connected, but not $\widehat{G}$, we might at least be able to construct a node-connected space-filling curve.

Due to the correspondence between graphs and grids, algorithms for graph partitioning are a frequently used tool for the partitioning of discretisation grids. As the graph partition problem, in general, is NP-complete, optimal solutions are too expensive to find. However, there are a couple of successful heuristic approaches, which are implemented by respective graph partitioning software (see the references section for a couple of examples).

11.2.1 The Edge Cut and the Surface of Partition Boundaries

As discussed in Sect. 10.1, the boundary (in 2D) or surface (in 3D) of a partition determines the amount of data to be communicated to neighbouring partitions. Hence, the surface-to-volume ratio of partitions is an important quality measure for partitions. In the graph setting, the surface of a partition is basically given by the size of the edge-cut of the dual graph (if we assume that cells have edges with a uniform length, such that each cut edge has the same contribution to the boundary length). For simple geometric objects, such as circles, spheres, squares, or cubes, the surface-to-volume ratio is proportional to $a^{(d-1)}/a^d$ for a d-dimensional object of extent a (diameter, side length, etc.). Zumbusch [285, 287] has shown that space-filling curves lead to partitions with the same asymptotic surface-to-volume ratio, if they are Hölder continuous and parameterised by volume (see also Exercise 11.4).

In practice, however, the length and shape of partition boundaries generated by space-filling curves are not absolutely optimal. Two examples – using the Peano curve to determine partitions on a uniformly refined and on an adaptive spacetree grid – are given in Fig. 11.1. The four partitions in the left image, for example, lead to 630 edges that are on a partition boundary – i.e., an edge cut of 630 (for the dual graph). The optimal solution for four partitions on this 81×81 grid is $4 \cdot 81 = 324$. On the bright side, the Peano curve would generate an optimal partitioning into nine partitions.

Graph partitioning algorithms that strive to minimize the edge cut when creating partitions of uniform size usually lead to results with shorter boundary surfaces. Again, the respective minimisation problem is known to be NP-complete, such that even solvers based on heuristics are computationally expensive. Partitioning based on space-filling curves can therefore be advantageous, if partitions have to be recomputed frequently, which is typically necessary for problems that require dynamic refinement and coarsening of the mesh. The asymptotically optimal surface-to-volume ratio assures that the partitions, which are guaranteed to be of the same size, will also be of a certain quality regarding surface lengths.

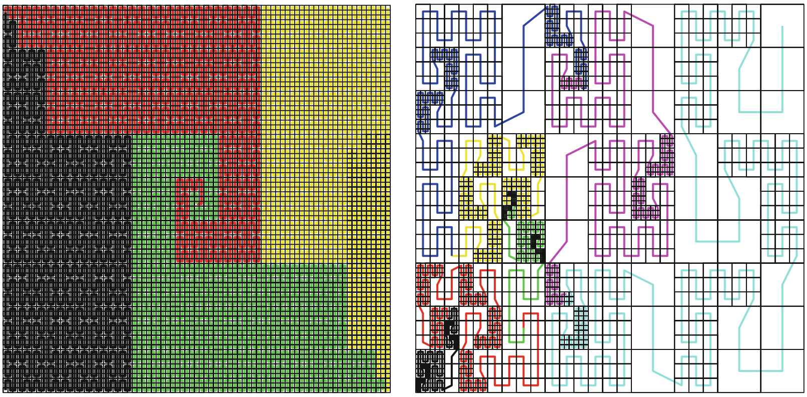
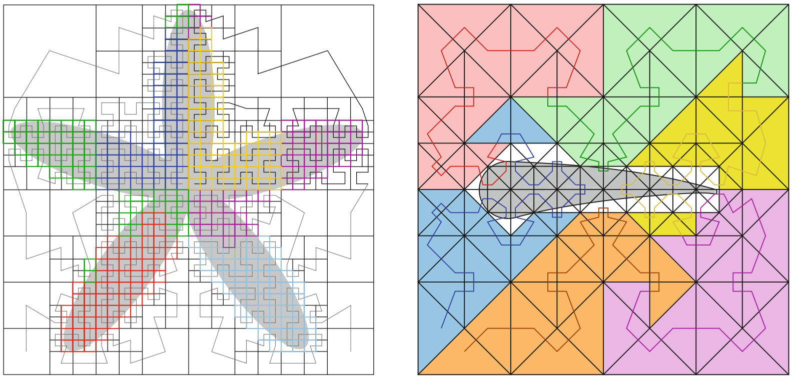


Fig. 11.1 Partitions defined via a Peano curve on a uniformly refined and on an adaptive spacetree grid



(a) Complicated domain geometry (b) Computational domain with an obstacle

Fig. 11.2 Disconnected partitions when using space-filling-curve partitioning

11.2.2 Connectedness of Partitions

For a connected space-filling curve, the partitions generated on the corresponding grid, as obtained from the recursive construction of the iterations, will be connected, as well. However, if we move to more complicated geometries, the respective grid partitions, even for connected space-filling curves, might no longer be connected. Two example scenarios are given in Fig. 11.2. In particular, disconnected partitions can occur due to the following reasons:

- A complicated computational domain is embedded into a unit square or cube, and the (possibly unstructured) discretisation grid is partitioned according to the

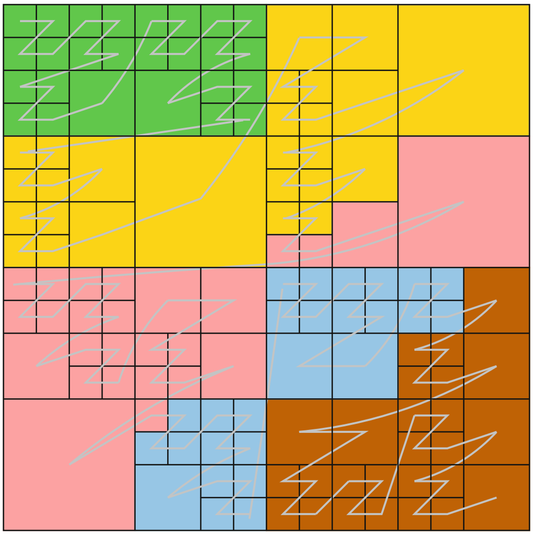


Fig. 11.3 An adaptive spacetree that is split into five partitions defined via Morton order

space-filling-curve indices of the grid cells. If the space-filling curve leaves and re-enters the domain, we can obtain a non-connected partition – as illustrated in Fig. 11.2a.

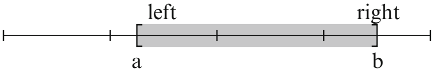
- Partitions can be disconnected by “holes” or obstacles in the domain, as illustrated in Fig. 11.2b.

Note that for both cases, partitions will not be disconnected, if we embed the computational domain (including holes and obstacles) in a spacetree or refinement-tree grid, and traverse the obstacle and boundary cells, as well (as it is common for marker-and-cell approaches). If such obstacle cells are coarsened as much as possible, the resulting overhead can usually be tolerated.

Connectedness of Partitions Defined by Morton Order

With *non-connected* space-filling curves, such as Morton order (or any order defined by the Lebesgue curve, the Z-curve, or respective variants), subsequent grid cells will not necessarily be direct neighbours. Hence, spacetree partitions defined via Morton order are not necessarily connected. On the other hand, partitions are higher-dimensional objects, so while a single cell of a spacetree partition might not be connected to its immediate neighbour in Morton order, it might still be connected to another spacetree cell of the same partition. Figure 11.3 gives an example of such a spacetree partitioning. Apparently, partitions will consist of at most two contiguous components. In the following, we will give a proof that this property is not restricted to the example in Fig. 11.3, but results from the special construction of the Morton order, and is thus valid for any spacetree partition, even if 3D or higher-dimensional Morton order is used.

Without loss of generality, we can restrict our discussion to regularly refined grids: the connectedness of a partition will not change, if we refine any grid cell of a spacetree to its deepest level L – assuming that subdivided cells will stay in the same partition as their parents, and making no assumption on the number of cells in a partition. We can thus assume that we have a grid size of $h = 2^{-L}$, and that the partitions are defined via the image cells of an interval $[a, b]$, where both a and b are multiples of 4^{-L}, i.e., they are boundaries of the nested intervals used in the well-known construction of Morton order. We can further assume that $|a - b| > 4^{-k}$ for a respective level k (using the largest possible k) and that $k \geq L$ (otherwise, move to level $L = k$), such that a and b are not in different nested intervals of size 4^{-k}. We thus have the following setting:

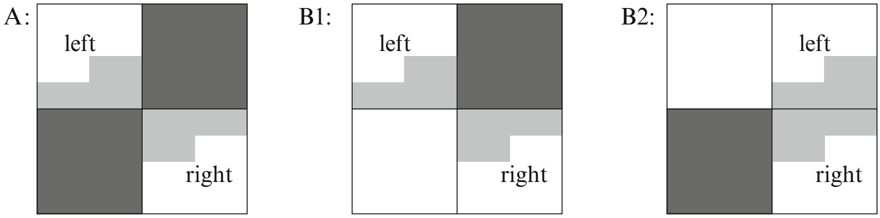


1. a is situated in a *left completion* interval of level k; all parameters larger than a are mapped to cells within the partition – we will call this part the left completion part of the partition.
2. b is situated in a *right completion* interval of level k; all parameters smaller than b are mapped to cells within the partition – we will call this part the right completion part of the partition.
3. Between the left and right completion intervals, we can have between 0 and 2 intervals of level k (note that we chose the largest possible k) that are entirely mapped to the considered partition, and which we shall call the centre part of the partition.

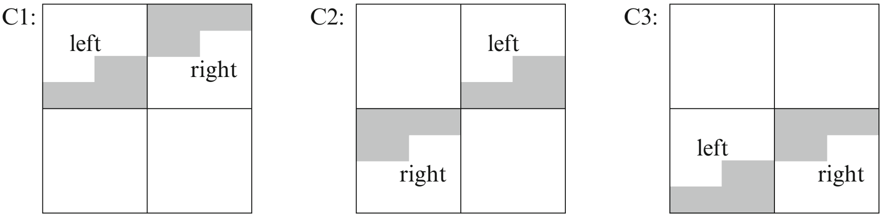
The left and right completion interval are both allowed to be empty, but neither of them should cover an entire level-k interval. If they are both empty, then the centre consists of at least one entire interval (we are not interested in empty partitions ...). If the left completion interval is not empty, it will contain the lower-right-most cell; the left completion part is thus connected to the lower and right neighbours within the centre parts of the partition. Similarly, the right completion part is connected to the upper and left neighbours within the centre parts.

This leads to only six possible scenarios how the resulting partitions can be arranged. These scenarios are illustrated in Fig. 11.4. The two tables in Fig. 11.4 summarise the number of connected parts of the partition, if we assume that both the left and the right completion parts are always connected. We see that we obtain at most two connected parts, where we consider parts that only touch at a single point as being not connected.

Hence, to finalise our proof, we need to show that left completion and right completion parts are indeed always connected. Figure 11.5 illustrates the possible scenarios, and it is easy to see that all four scenarios lead to a connected partition, if the smaller left completion part is again connected. Hence, we obtain a recursion argument, which we can easily prove by induction over the refinement level. For the connectedness of the right completion part, we can generate an analogous proof



scen.	left compl.	right compl.	partitions	comment
A	empty	empty	2	parts 2 and 3 touch
	left or right not empty		1	parts 2 and 3 connected by last cell of left or first cell of right completion part
B1	∗	non-empty	1 or 2	left connected to part 2; right not necessarily connected to left completion
	∗	empty	1	left connected to part 2
B2	non-empty	∗	1 or 2	right connected to part 3; left not necessarily connected to right completion
	empty	∗	1	right connected to part 3



scen.	left compl.	right compl.	partitions	comment
C1	∗	∗	1 or 2	left not necessarily connected to right completion
C2	∗	∗	1 or 2	left not necessarily connected to right completion
C3	∗	∗	1 or 2	left not necessarily connected to right completion

Fig. 11.4 Possible connectedness scenarios of partitions defined via Morton order. '∗' marks a "don't care" case (empty or non-empty)

by induction, such that our proof is finished: partitions defined via a Morton order mapping of a parameter interval consist of at most two connected parts. Note that the property and also the general idea of the proof carry over to 3D and to higher-dimensional cases.

References and Further Readings

Locality measures, such as in Eqs. (11.1) and (11.2) were established by Gotsman and Lindenbaum [105]. who proved lower and upper bounds for the Hilbert curve in the Euclidean metric. Niedermeier et al. [196] introduced a framework to prove

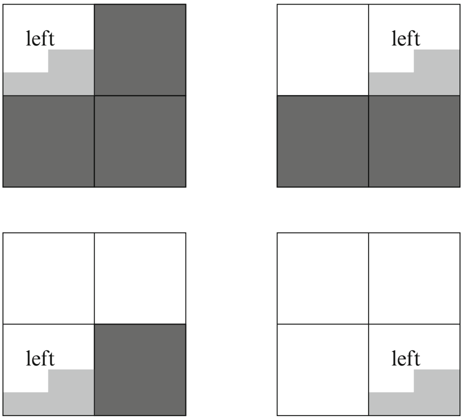


Fig. 11.5 Connectedness scenarios for the *left* completion part

tighter bounds, and presented respective bounds for the 2D Hilbert curve that come close to the lower bound of $\tilde{C}_2 = 6$. They also provided locality coefficients for the Hilbert curve in the Manhattan metrics, and for the H-index (in Euclidian, Manhattan, and maximum metrics). The results for 3D Hilbert curves, as given in Table 11.1, are also obtained from [196] (see also their review on further work on this topic). A proof that the dilation factor of the 2D Hilbert curve is exactly 6 was given by Bauman [34].

Partition-Oriented Locality Measures

The Hölder continuity and the respective locality measures discussed in Sect. 11.1 actually relate the area of a partition to the distance of the image points of the interval boundaries that define the partition. However, if space-filling curves are used to define the partitions on a computational grid, we would rather like to relate the area of a partition (which reflects the number of grid points) to the diameter or largest extent of the partition. Or similar, we would like to know the ratio between the length of the partition boundary and its area. Zumbusch [285] showed that the quasi-optimality of the Hölder continuity does transfer to the boundary-to-area (or surface-to-volume) ratio of partitions. Respective discrete locality measures were also studied by Hungershöfer and Wierum [135, 268], who computed the surface-to-area and diameter-to-area ratios of partitions induced by a space-filling curve in comparison to the respective ratios of a square. Tirthapura et al. [256] gave an analysis of the number of remote accesses (to nearest neighbour cells), if partitions are generated via space-filling curves. A comparison of the partition boundaries and edge-cuts of space-filling-curve partitions vs. graph-based partitioners applied to different finite element grids was given by Schamberger and Wierum [239].

Haverkort and Walderveen [123] proposed locality measures that are motivated by using space-filling curves to create bounding boxes for point data. In their work, they also provide a review on existing results regarding various locality measures for space-filling curves.

"Inverse" Locality

While Hölder locality quantifies the quality of the mapping indexing function $\mathcal{I} \to \mathcal{Q}$, or $\{1,\dots,n^d\} \to \{1,\dots,n\}^d$ in the discrete case, the locality of an indexing $\bar{c}\colon \{1,\dots,n\}^d \to \{1,\dots,n^d\}$ is also an interesting question (it is impossible to define a continuous mapping $\mathcal{Q} \to \mathcal{I}$). Mitchison and Durbin [190] proposed

$$\sum \left(|\bar{c}(i+1,j) - \bar{c}(i,j)|^q + |\bar{c}(i,j+1) - \bar{c}(i,j)|^q\right)$$

as a locality measure for 2D array indexings, where q penalises (if $q > 1$) or tolerates ($q < 1$) large distances between adjacent cells. For $q \leq 1$, optimal indexings have to be ordered, i.e., monotonous in the coordinate directions, which means that the standard space-filling curve (excluding the Lebesgue curve) indexings cannot be optimal.

Even though indexings based on space-filling curves will usually not map a square partition to a single contiguous index sequence, we usually get by with only few such sequences – especially, if we allow a certain overhead of "unused" indices. When designing data structures based on space-filling curves, it is an interesting question how many contiguous index sequences are required to cover a partition. Jagadish [139, 140] and Moon et al. [191] studied this question in the context of data bases (see also the references in these articles). Asano et al. [18] discussed it for range queries for image data bases, and introduced a special 2D space-filling curve that only requires three contiguous index sequences for any partition rectangle, if a certain overhead is allowed. Interestingly, the locality coefficients C_α of their curve are slightly weaker than the Hilbert curve (for Euclidian and Manhattan measure, see [123]). In such data base applications, covering partitions by fewer contiguous sequences leads to a substantial speed-up, because contiguous data can be read very fast compared to the time required to search a new partition start.

Note that the question how many contiguous index sequences cover a given domain is related to the question of connected partitions: If, in Fig. 11.2a, we consider the five "petals" of this flower-shaped domain as separate domains that are covered by contiguous sequences, we see that every petal is likely to be covered by multiple index partitions.

What’s next?

- The next chapter will take up the discussion on Sierpinski curves in 2D and 3D. The question of locality properties, in this case, is strongly connected with the shape of the triangles and tetrahedra generated by the bisection process.
- If you want to skip the triangular and tetrahedral constructions, you can on with Chap. 13, which will discuss the effect of locality properties on the cache efficiency of resulting algorithms.

Exercises

11.1. Examine whether the Lebesgue curve is Hölder continuous.

11.2. Compute the diameter-to-volume ratios for simple geometrical objects in 2D and 3D, such as circles, squares, cuboids, spheres, etc., and compare these ratios with the respective constants obtained for the Hölder continuity (as given in Table 11.1).

11.3. Write a simple test program that generates random pairs of indices and computes the ratio between index and point distance for a given space-filling curve (using respective algorithms to compute the image point from a given index). Use this program to test the coefficients given in Table 11.1.

11.4. Use a case study similar to that in Sect. 11.2.2 to compute an upper bound for the surface-to-volume ratio of partitions obtained by Hilbert order. Discuss whether a similar upper bound can also be obtained for the Lebesgue curve.

11.5. If we examine the computation of Hölder coefficients for space-filling curves, we notice that sequences of consecutive squares (i.e., adjacent squares that are sequentialised by a straight line segment) are, in general, disadvantageous. Try to generate iterations on small $k \times k$ grids, and try to keep the longest “straight” sequence of cells as small as possible. Pay attention to what happens, if these basic patterns are to be repeated in order to construct higher iterations.

Chapter 12
Sierpinski Curves on Triangular and Tetrahedral Meshes

Triangular and tetrahedral grids are widely used in numerical simulation. Especially in the "world" of Finite Element methods, triangles and tetrahedra are the most popular building blocks for computational grids – regardless of whether such grids are unstructured or structured, or whether they are uniform or adaptive. Defining sequential orders on such grids – for parallelisation, efficient data structures, or just to improve performance – is therefore a well-studied task (see the references section of this chapter). A straightforward approach to compute such orderings could be to compute the indices of the centre points of grid cells, for example, using an arbitrary space-filling curve. However, such an approach will not take advantage of any structure that was imposed during grid generation. In this chapter, we will therefore discuss variants of the Sierpinski curve in two and three dimensions that are constructed in a compatible way for an underlying (structured) triangular or tetrahedral grid.

12.1 Triangular Meshes and Quasi-Sierpinski Curves

12.1.1 Triangular Meshes Using Red–Green Refinement

A frequently used method for structured refinement of triangular grids is called *red–green refinement*, and is illustrated in Fig. 12.1. In each regular refinement step, a triangle is subdivided into four child triangles. The vertices of the new triangles are located on the centres of the three edges. As we can see from Fig. 12.1, the resulting child triangles have identical side lengths and are therefore congruent to each other. Note that the actual side length a, b, and c of the triangles can be arbitrary – hence, we obtain a mesh of congruent triangle cells for an arbitrary root triangle. The respective refinement step is referred to as *red refinement*.

Using such regular refinement steps only, adaptive grids will necessarily contain so-called *hanging nodes*, i.e., nodes that are vertices in some of the adjacent

M. Bader, *Space-Filling Curves*, Texts in Computational Science and Engineering 9,
DOI 10.1007/978-3-642-31046-1_12, © Springer-Verlag Berlin Heidelberg 2013

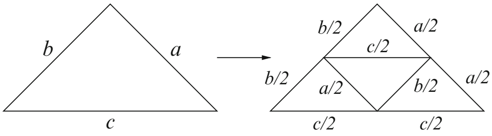


Fig. 12.1 Basic red–green refinement scheme – "red" refinement

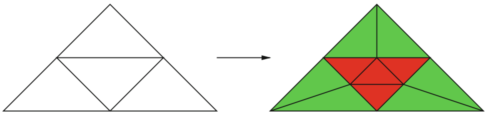


Fig. 12.2 Basic red–green refinement scheme – "green" refinement

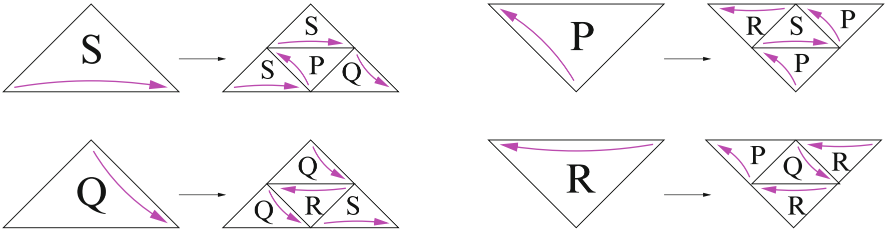


Fig. 12.3 Recursive construction and resulting grammar for the construction of a node-connected quasi-Sierpinski curve on triangular meshes resulting from uniform red–green refinement

triangles, but lie on an edge of another adjacent triangle. As such hanging nodes require separate treatment in numerical schemes, they are often avoided during grid generation. For triangular cells, this is easily solved via bisection of the triangles at the hanging node, as illustrated in Fig. 12.2. The corresponding refinement step is referred to as *green refinement*. If we require further refinement of a "green" triangle, though, we should turn it into a "red triangle" first, and proceed with "red refinement" on this grid. Hence, the two children of a green triangle should not be split into smaller triangles in order to avoid distortion of the triangular cells.

12.1.2 Two-Dimensional Quasi-Sierpinski Curves

From the refinement scheme in Fig. 12.1, we can see that it will not be possible to construct a Sierpinski curve in the regular manner. More specifically, we cannot construct an *edge-connected* curve. The reason is simply that the three child triangles that are situated in the corners have only one edge-connected neighbour, which is the central child triangle for all three. Hence, a contiguous path that connects all four triangles via common edges does not exist.

However, we can construct an order on the child triangles such that they are *node-connected*. The respective recursive construction scheme is given in Fig. 12.3. The building patterns, denoted by S, P, Q, and R in Fig. 12.3 can again be turned into a grammar to construct iterations or approximating polygons of a potential space-filling curve.

Algorithm 12.1: Computing the n-th approximating polygon of a quasi-Sierpinski curve

```
Procedure quasisierp(x1, x2, x3, n)
    Parameter: x1,x2,x3: vertices (as 2D coordinates); n: refinement level
    Data: curve: list of vertices (empty at start, contains iteration points on exit)
begin
    if $n > 0$ then
        return attach(curve, x1, x2)
    else
        quasisierp(x1, mid(x1,x2), mid(x1,x3), n-1);
        quasisierp(mid(x1,x2), mid(x1,x3), mid(x2,x3), n-1);
        quasisierp(mid(x1,x3), mid(x2,x3), x3, n-1);
        quasisierp(mid(x2,x3), x2, mid(x1,x2), n-1);
    end
end
```

An alternative and simpler implementation to construct the red–green refinement scheme, however, follows the notation we introduced in Sect. 6.2 for generalised Sierpinski curves. As we no longer have a tagged edge bisection, we will denote triangles as triples (x_1, x_2, x_3) of the vertices x_1, x_2, and x_3 (i.e., using parentheses instead of square brackets). The new vertices of the child triangles are then given by $\frac{1}{2}(x_1 + x_2)$, $\frac{1}{2}(x_1 + x_3)$, and $\frac{1}{2}(x_2 + x_3)$, and we can formulate the refinement rule as follows:

$$(x_1, x_2, x_3) \to \begin{array}{l} \left(x_1, \frac{x_1+x_2}{2}, \frac{x_1+x_3}{2}\right), \left(\frac{x_1+x_2}{2}, \frac{x_1+x_3}{2}, \frac{x_2+x_3}{2}\right), \\ \left(\frac{x_1+x_3}{2}, \frac{x_2+x_3}{2}, x_3\right), \left(\frac{x_2+x_3}{2}, x_2, \frac{x_1+x_2}{2}\right). \end{array} \tag{12.1}$$

In this notation, the order of the vertices in the triples (x_1, x_2, x_3) is always chosen such that x_1 is the vertex, where the curve enters, and x_2 is the vertex where the curve leaves the triangle. Hence, the edge x_1x_2 is part of the approximating polygon of the resulting curve. The approximating polygon can thus be generated by recursively generating the triangles as given by Eq. (12.1), and then connecting the first two edges in each triangle. Such an implementation via vertex labelling is given in Algorithm 12.1.

Quasi-Sierpinski Curves

The construction given by Eq. (12.1) and Algorithm 12.1 defines a space-filling curve that fills the given initial triangle (x_1, x_2, x_3). Instead of giving a rigorous definition of the curve's mapping and a full proof that this mapping is continuous and space-filling, we will just summarise the most important properties:

- In the usual way, we will map intervals of length 4^{-n} (n the refinement level) to triangles, which are constructed by recursive quadrisection.

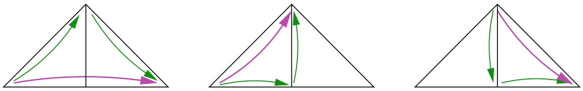


Fig. 12.4 Local ordering for green refinement of triangular cells

- In each quadrisection step, the side lengths of the triangles are halved. As all triangles on the same refinement level are congruent, the longest side length of all triangles will be halved in each refinement step.
- Hence, two parameters t_0 and t_1 with distance less than 4^{-n} will be located in neighbouring intervals, and will be mapped to two node-connected triangles with maximum side lengths $c \cdot 2^{-n}$. The distance of the image points is therefore $2c \cdot 2^{-n}$, at most.
- Comparing the proofs for continuity and Hölder continuity of the Hilbert and Peano curve, we can infer that our quasi-Sierpinski curve is Hölder continuous with exponent $\frac{1}{2}$.

According to our classification introduced in Sect. 7.1, quasi-Sierpinski curves are recursive, node-connected space-filling curves. We can infer from our discussion that such curves are obviously Hölder continuous with exponent d^{-1}, in general.

12.1.3 Red–Green Closure for Quasi-Sierpinski Orders

Quasi-Sierpinski curves are based on uniform, red-refinement grids. If we switch to refinement-tree approaches, as introduced in Chap. 9, we will obtain structured adaptive triangular grids similar to those introduced in Sect. 9.3. If we want to avoid hanging nodes, and thus require green refinement, we have to define local orders on the two child cells of a green triangle. Such orders for the three possible combinations of entry- and exit-vertices are given in Fig. 12.4. As green refinement stops after one refinement level, and is replaced by red refinement for further refinement, we do not need to consider whether these local orders can be extended towards a closed, nested recursive scheme – though, it wouldn't be too difficult.

12.2 Tetrahedral Grids and 3D Sierpinski Curves

12.2.1 Bisection-Based Tetrahedral Grids

In Sect. 8.3, we already discussed a 3D version of the Sierpinski curve. However, it turned out that this Sierpinski curve has much weaker locality properties than what we would like to have. Most importantly, the ratio between the side lengths of the generated tetrahedral cells is not bounded, such that we obtain heavily distorted cells. As a result, the corresponding Sierpinski curve is no longer Hölder continuous

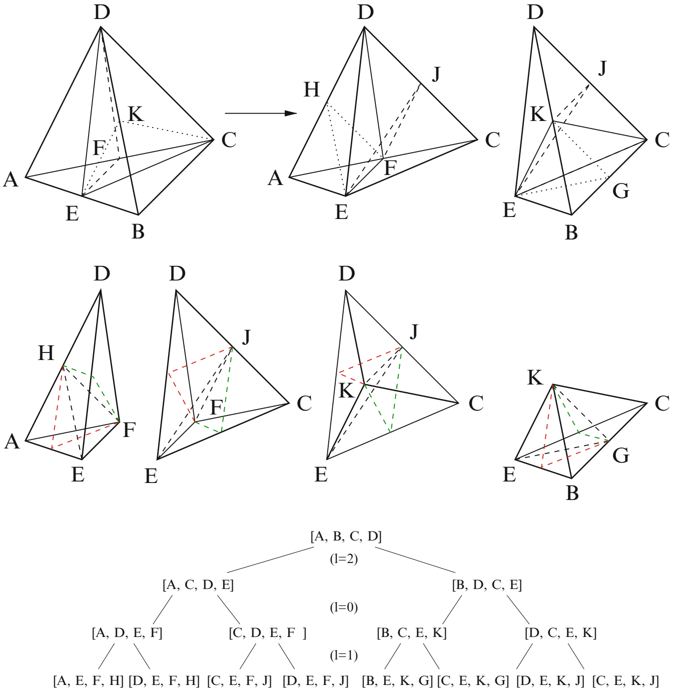


Fig. 12.5 Tetrahedral refinement (generated tetrahedra and 4-tuple notation) according to the refinement rules given in Eq. (12.2). Note that we start with the case $l = 2 \pmod 3$

with exponent $d^{-1} = \frac{1}{3}$. Hölder continuity can still be shown, but with a weaker exponent (see the solution of Exercise 12.2).

For many applications, it is important, though, that certain locality properties or shape restrictions of the generated cells are preserved. Hence, we require a generation scheme that is able to preserve the tetrahedral shapes. We will restrict ourselves to bisection-based refinement and also retain our 4-tuple notation with the tagged-edge convention, at first. Then, the following refinement scheme was shown to generate properly shaped tetrahedra. It uses a different refinement rule on every third refinement level l:

$$\begin{aligned}
[x_1, x_2, x_3, x_4] &\xrightarrow{l=0 \pmod 3} [x_1, x_3, x_4, x_s], [x_2, x_3, x_4, x_s] \\
[x_1, x_2, x_3, x_4] &\xrightarrow{l=1 \pmod 3} [x_1, x_3, x_4, x_s], [x_2, x_3, x_4, x_s] \\
[x_1, x_2, x_3, x_4] &\xrightarrow{l=2 \pmod 3} [x_1, x_3, x_4, x_s], [x_2, x_4, x_3, x_s]
\end{aligned} \tag{12.2}$$

where $x_s = \frac{1}{2}(x_1 + x_2)$ is the new vertex on the tagged edge $x_1 x_2$. Figure 12.5 illustrates the tetrahedra that are generated by this splitting scheme.

Local and Global Refinement Edges

As a key feature of the introduced scheme for tetrahedral refinement, it turns out that the faces of the generated tetrahedra are bisected according to the classical 2D scheme. We can therefore identify a *local refinement edge* for each face of a tetrahedron. Once this face is split during one of the following bisection steps, this local refinement edge will carry the new vertex. After the split, the two other edges will become the local refinement edges in their respective triangular face, just as in the 2D construction. In any tetrahedral cell produced by this construction, two of the four faces will have the same local refinement edge. In our refinement scheme (12.2), the tetrahedra are always split at this edge, which we therefore call a *global refinement edge*. Figure 12.6 illustrates the local and global refinement edges for our refinement scheme.

In Fig. 12.6, we also introduce the so-called red–black classification of the tetrahedra in the present refinement scheme. A tetrahedron is called *black*, if the two other local refinement edges (i.e., those that are not global refinement edges) share a common vertex. Otherwise, they are called red. The red–black classification was used by Bänsch [43], who introduced the respective refinement scheme directly via different refinement rules for red and black tetrahedra. He used the red–black classification of the current tetrahedron and its parent to determine the position of the local refinement edges in the bisected child tetrahedra. The refinement scheme given by Eq. (12.2) was introduced by Kossaczky [151] as a simpler description of Bänsch's refinement algorithm. Figure 12.6 thus illustrates the equivalence of the two schemes, which we will refer to as the Bänsch-Kossaczky scheme. It also shows that the global refinement edges are unique in the sense that there cannot be two pairs of adjacent local refinement edges.

On the Shapes of the Generated Tetrahedra

Liu and Joe [165], who introduced a closely related bisection scheme for tetrahedra (see the references section), used the tetrahedron given in Fig. 12.7 to derive the shapes of the bisected tetrahedra. Their so-called *canonical* tetrahedron is given by

$$T_{\text{can}} := [(-1,0,0),(1,0,0),(0,0,1),(0,\frac{1}{2}\sqrt{2})]. \qquad (12.3)$$

It is easily computed (see Exercise 12.4) that the Bänsch-Kossaczky scheme, when applied to this canonical tetrahedron, will generate congruent tetrahedra on each bisection level. The two tetrahedra of the first level, for example, result from a bisection at edge $((-1,0,0),(1,0,0))$ and are symmetric to the yz-plane (compare Fig. 12.7). After three bisection steps, the resulting eight tetrahedra are all congruent to each other, and are scaled-down versions of the canonical tetrahedron.

If we compute a mapping γ that maps an arbitrary tetrahedron $[A,B,C,D]$ to T_{can}, then γ will map any children, grandchildren, etc. of $[A,B,C,D]$ to

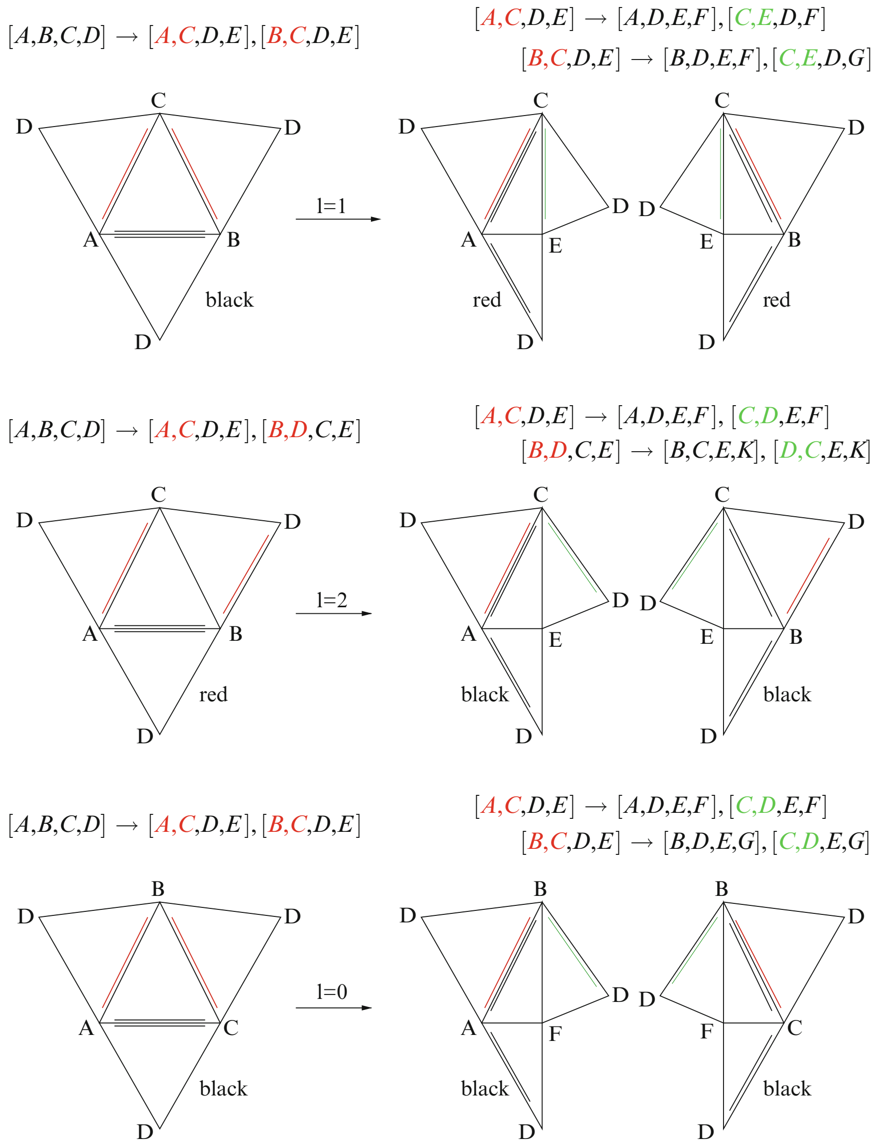

$[A,B,C,D] \rightarrow [A,C,D,E],[B,C,D,E]$
$[A,C,D,E] \rightarrow [A,D,E,F],[C,E,D,F]$
$[B,C,D,E] \rightarrow [B,D,E,F],[C,E,D,G]$
C D D A B D black
l=1
C D D A E red D
C D D E B red D
$[A,B,C,D] \rightarrow [A,C,D,E],[B,D,C,E]$
$[A,C,D,E] \rightarrow [A,D,E,F],[C,D,E,F]$
$[B,D,C,E] \rightarrow [B,C,E,K],[D,C,E,K]$
C D D A B red D
l=2
C D D A E black D
C D D E B black D
$[A,B,C,D] \rightarrow [A,C,D,E],[B,C,D,E]$
$[A,C,D,E] \rightarrow [A,D,E,F],[C,D,E,F]$
$[B,C,D,E] \rightarrow [B,D,E,G],[C,D,E,G]$
B D D A C black D
l=0
B D D A F black D
B D D F C black D

Fig. 12.6 Local and global refinement edges for the bisection scheme (12.2). The tetrahedra are shown cut open at the three edges that meet at a vertex opposite to the refinement edge. The resulting *red–black* classification is derived for all tetrahedra

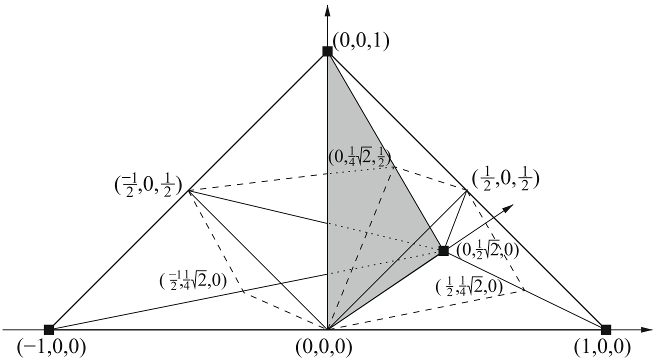


Fig. 12.7 The canonical tetrahedron given by Liu and Joe [165] (vertices marked by *boxes*) – given are the coordinates of the split points of the first three bisection steps. Note the symmetry of the two child tetrahedra (symmetric to the highlighted yz-plane)

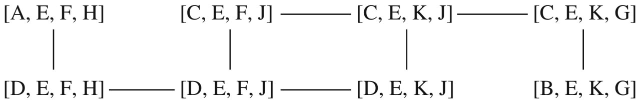


Fig. 12.8 Neighbouring tetrahedra after three bisection steps of the Kossaczky scheme (12.2), starting with level $l = 2$ (compare Fig. 12.5). The graph connects tetrahedra, if they share a common face. Obviously there is no face-connected iteration that connects all eight tetrahedra

corresponding children and grandchildren of T_{can}. As a result, the shapes of the tetrahedra in a bisection-generated mesh will fall into a limited set of congruency classes. In particular, their shapes will not degenerate, such that, for example, the ratio between longest and shortest edge stays bounded, but also the interior angles of the tetrahedra stay within a certain range. The latter property is essential for the use of tetrahedral meshes in many applications of scientific computing, such as for Finite Elements.

12.2.2 Space-Filling Orders on Tetrahedral Meshes

We would, of course, like to carry over the construction of the 3D Sierpinski curve presented in Sect. 8.3 to the tetrahedral meshes constructed via the Bänsch-Kossaczky scheme. The result would be a face-connected space-filling curve, which should then be Hölder continuous with exponent $\frac{1}{3}$. Figure 12.8, however, shows that such a construction cannot exist. In the respective graph, the eight tetrahedra generated after three bisection steps (starting with a *red* tetrahedron) form the nodes, and are connected by graph-edges, if they share a common face. A face-connected Sierpinski iteration on the given eight child tetrahedra would thus be equivalent to a contiguous path that connects all eight children. It is easy to check that such a path unfortunately does not exist. We therefore cannot construct a face-connected 3D space-filling curve following this approach.

A Three-Dimensional, Node-Connected Quasi-Sierpinski Curve

It is possible, however, to construct a node-connected, quasi-Sierpinski curve on grids generated from the Bänsch-Kossaczky scheme. In principle, we only have to track the entry and exit points of the Sierpinski curve within the tetrahedral (parent) cells, which leaves quite a lot of choices for the construction. In the following, we place the following restrictions:

- Our choice shall be purely local: for each parent tetrahedron, we want to determine a first and second child, such that a depth-first traversal that obeys this local order will produce the grid traversal.
- As candidates for the entry and exit vertices of the curve in the child cells, we have the three vertices at the common face. The new vertex that is introduced at the split edge is not immediately used as entry or exit point of the curve in the respective child tetrahedra. If both other vertices remain as candidates (only, if entry and exit vertex of the parent cell are both adjacent to the split edge), we chose the vertex with the smaller index in the 4-tuple, i.e., the "oldest" possible vertex.
- If two scenarios are identical up to the entry and exit vertices being switched, we use the same entry/exit vertex in the children. Thus, the resulting curves just change orientation.
- As the refinement rules for $l = 0$ and $l = 1$ are identical in the Kossaczky scheme, we try to keep these rules identical also with respect to the entry and exit vertices.

The resulting refinement scheme is given in Eqs. (12.4) and (12.5) – for each tetrahedron, the entry vertex is marked by a bar below the vertex; exit vertices are marked by bars above the vertex:

- for $l = 0 \pmod 3$ or $l = 1 \pmod 3$:

$$\begin{aligned}
\left[\underline{x_1}, \overline{x_2}, x_3, x_4\right] &\longrightarrow \left[\underline{x_1}, \overline{x_3}, x_4, x_s\right], \left[\overline{x_2}, \underline{x_3}, x_4, x_s\right] \\
\left[\overline{x_1}, \underline{x_2}, x_3, x_4\right] &\longrightarrow \left[\underline{x_2}, \overline{x_3}, x_4, x_s\right], \left[\overline{x_1}, \underline{x_3}, x_4, x_s\right] \\
\left[\underline{x_1}, x_2, \overline{x_3}, x_4\right] &\longrightarrow \left[\underline{x_1}, x_3, \overline{x_4}, x_s\right], \left[x_2, \overline{x_3}, \underline{x_4}, x_s\right] \\
\left[\overline{x_1}, x_2, \underline{x_3}, x_4\right] &\longrightarrow \left[x_2, \underline{x_3}, \overline{x_4}, x_s\right], \left[\overline{x_1}, x_3, \underline{x_4}, x_s\right] \\
\left[x_1, \underline{x_2}, \overline{x_3}, x_4\right] &\longrightarrow \left[\underline{x_2}, x_3, \overline{x_4}, x_s\right], \left[x_1, \overline{x_3}, \underline{x_4}, x_s\right] \\
\left[x_1, \overline{x_2}, \underline{x_3}, x_4\right] &\longrightarrow \left[x_1, \underline{x_3}, \overline{x_4}, x_s\right], \left[\overline{x_2}, x_3, \underline{x_4}, x_s\right]
\end{aligned} \tag{12.4}$$

- for $l = 2 \pmod 3$:

$$\begin{aligned}
\left[\underline{x_1}, \overline{x_2}, x_3, x_4\right] &\longrightarrow \left[\underline{x_1}, \overline{x_3}, x_4, x_s\right], \left[\overline{x_2}, x_4, \underline{x_3}, x_s\right]\\
\left[\overline{x_1}, \underline{x_2}, x_3, x_4\right] &\longrightarrow \left[\underline{x_2}, \overline{x_4}, x_3, x_s\right], \left[\overline{x_1}, x_3, \underline{x_4}, x_s\right]\\
\left[\underline{x_1}, x_2, \overline{x_3}, x_4\right] &\longrightarrow \left[\underline{x_1}, x_3, \overline{x_4}, x_s\right], \left[x_2, \underline{x_4}, \overline{x_3}, x_s\right]\\
\left[\overline{x_1}, x_2, \underline{x_3}, x_4\right] &\longrightarrow \left[x_2, \overline{x_4}, \underline{x_3}, x_s\right], \left[\overline{x_1}, x_3, \underline{x_4}, x_s\right]\\
\left[x_1, \underline{x_2}, \overline{x_3}, x_4\right] &\longrightarrow \left[\underline{x_2}, \overline{x_4}, x_3, x_s\right], \left[x_1, \overline{x_3}, \underline{x_4}, x_s\right]\\
\left[x_1, \overline{x_2}, \underline{x_3}, x_4\right] &\longrightarrow \left[x_1, \underline{x_3}, \overline{x_4}, x_s\right], \left[\overline{x_2}, \underline{x_4}, x_3, x_s\right]
\end{aligned} \tag{12.5}$$

The scheme may be implemented as a vertex-labelling algorithm, similar to Algorithm 8.3 for our face-connected curve (with bad locality). However, according to the refinement rules in (12.4) and (12.5), the two recursive calls will occur in different order depending on the refinement level and on the current location of entry and exit points. This information can be coded in a respective refinement table, which is easily obtained from (12.4) and (12.5). Figure 12.9 plots the resulting curve, if the unit tetrahedron $[(0,0,0),(1,0,0),(0,1,0),(0,0,1)]$ is used as root domain.

Hölder Continuity of the Generated Curve

Our 3D quasi-Sierpinski curve is not strictly recursive and only node-connected, such that we cannot directly infer Hölder continuity for the respective mapping s. However, we can exploit the fact that the tetrahedral cells belong only to a limited number of different congruency classes. Hence, even for an arbitrary initial tetrahedron, we know that after $3k$ bisections, the longest edge of any tetrahedron on this level will be bounded by $c \cdot \frac{1}{2^k}$, where c is a constant that is determined by the "worst" congruency class.

Assume that the mapping s is defined via the usual construction, using nested intervals that are split in eight intervals in each recursion step, and, correspondingly, direct refinement of the tetrahedral subdomains in eight children. On each level n, the edge lengths of the tetrahedra are then bounded by $c \cdot \frac{1}{2^n}$, and the proof of Hölder continuity is straightforward:

- For any choice of two parameters t_1 and t_2, pick an n, such that $8^{-(n+1)} \leq |t_1 - t_2| < 8^{-n}$. Thus, t_1 and t_2 will be in the same or in adjacent intervals.
- Their image points $s(t_1)$ and $s(t_2)$ will, in the worst case, be located in two tetrahedra of level n that share at least a common vertex.
- As the longest edge of these two tetrahedra is bounded by $c \cdot \frac{1}{2^n}$, the distance of the image points will be $\|s(t_1) - s(t_2)\| \leq 2c\frac{1}{2^n}$.
- After short computation, we obtain that $\|s(t_1) - s(t_2)\| \leq 4c|t_1 - t_2|^{1/3}$.

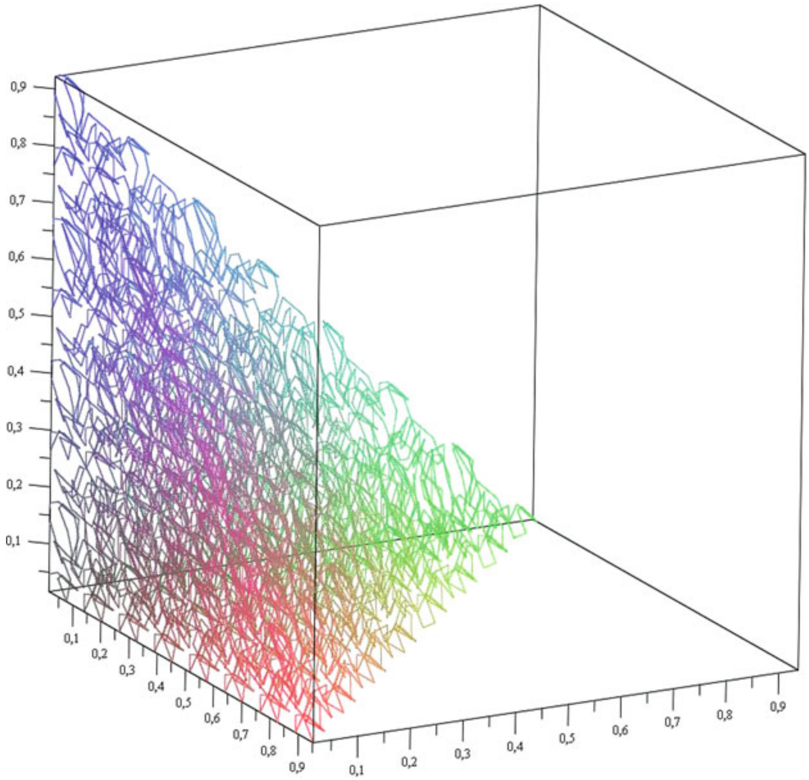


Fig. 12.9 The 12-th iteration of the node-connected quasi-Sierpinski curve generated by the scheme given in (12.4) and (12.5)

Hence, the 3D node-connected quasi-Sierpinski curve is Hölder continuous with exponent $\frac{1}{3}$. It is also straightforward to prove that the curve is parameterised by volume, such that we can compute parallel partitions using this curve, and infer all compactness properties discussed in Chap. 10.

References and Further Readings

Tetrahedral Bisection

Mesh refinement schemes that are based on bisection of tetrahedral cells have been proposed by several authors. In Sect. 12.2,we discussed the scheme introduced by Bänsch [43], following the notation Kossaczky [151] developed for this scheme. The tetrahedral mesh generation process by Maubach [176] is also based on the tuple notation (x_1, x_2, x_3, x_4) of a tetrahedron, with vertices $x_1, x_2, x_3, x_4 \in \mathbb{R}^3$. Similar to the Bänsch-Kossaczky scheme, it uses different bisection edges depending on the refinement level l. With $k := 3 - (l \pmod 3)$, Maubach defines bisection on level l via the rule

$$[x_1, x_2, x_3, x_4] \to [x_1, \dots, x_k, x_s, x_{k+2}, \dots, x_4], [x_2, \dots, x_{k+1}, x_s, x_{k+2}, \dots, x_4], \tag{12.6}$$

where $x_s = \frac{1}{2}(x_1 + x_{k+1})$. Note that the sequences $x_1, \dots, x_k$ and $x_{k+1}, \dots, x_4$, as well as $x_2, \dots, x_{k+1}$ or $x_{k+2}, \dots, x_4$ might be empty or consist of only one vertex. Exercise 12.5 deals with the question of equivalence of the two schemes. A symbolic refinement algorithm for the Maubach scheme was also given by Hebert [125].

Further bisection-based schemes for tetrahedral meshes were introduced by Arnold et al. [17], and by Liu and Joe [165, 166]. Both schemes are defined on a type classification of tetrahedra – two types that correspond to Bänsch's red/black classification plus two additional types. Arnold et al. [17] also discussed the similarities and differences of the four schemes: the schemes are essentially equivalent, if meshes that are build from a single initial tetrahedral cell are studied, and if this tetrahedron is black or red according to Bänsch's scheme. Liu and Joe [165] used the canonical tetrahedron given in Fig. 12.7 to prove that their scheme produces tetrahedra that fall into a limited set of congruency classes. Due to the similarity of the schemes, this result can be essentially transferred to the other three schemes. Liu and Joe also proved that the diameters $\delta(T_n)$ of tetrahedra T_n on the n-th bisection level will be smaller than $c(1/2)^{n/3}\delta(T)$, where T is the initial tetrahedron [165]. Together with the node-connectedness, this result directly leads to Hölder continuity of the quasi-Sierpinski curve discussed in Sect. 12.2.2.

A different bisection scheme was introduced by Rivara and Levin [227], who always use the longest edges of tetrahedra as refinement edges (longest-edge bisection). It is similar to the scheme of Liu and Joe [165] in the sense that Liu and Joe applied longest edge bisection to the canonical tetrahedron given in Fig. 12.7, which is then mapped to tetrahedral cells to transfer the bisections to the respective cells. Hence, while the schemes may produce similar sequences of bisections, they will usually lead to different meshes.

Sierpinski Orders in 3D

The inadequate bisection refinement discussed in Sect. 8.3, as well as the corresponding Sierpinski curve, seems to be an example for the problem that "failures" are not too often published: there seems to be no discussion that quantifies the problems with the simple refinement schemes, and my literature search on 3D Sierpinski curves brought up only a single paper that actually used this curve: Pascucci [211] uses a respective order on tetrahedral grids to speed up isosurface computation (see references on triangular and tetrahedral stripping in Chap. 14). He also reported that the curve had apparently not been described before.

The use of generalised, (quasi-)Sierpinski curves for parallelisation of problems discretised on tetrahedral grids was established by Zumbusch [287]. Mitchell [188] discusses the construction of quasi-Sierpinski orders on both triangular and tetrahedral meshes that stem from a structured refinement approach. He also discussed quadrilateral refinement schemes. His article on the refinement-tree partitioning method [189] is the basis for the scheme presented in Sect. 12.2.2 – however, for our Sierpinski curve, we avoided to use vertices on bisection edges as entry and exit vertex in the child cells.

Hamiltonian Paths and Self-Avoiding Walks

To compute a space-filling order on an unstructured grid is equivalent to finding an Hamiltonian Path through the grid's dual graph. If the respective paths do not intersect themselves, they are referred to as *self-avoiding walks*. To transfer the typical space-filling-curve construction to unstructured grids has been attempted by a couple of authors. Heber et al. [124], for example, examined node-connected orders on structured adaptive triangular grids in the context of self-avoiding walks. Schamberger and Wierum [238] exploited the structure of modified refinement trees to infer so-called *graph-filling curves*.

What's next?

- ➔ The next two chapters will provide case studies on how to exploit the properties induced by space-filling curves for different aspect of algorithms in scientific computing. Chap. 13 will focus on matrix operations, while Chap. 14 will deal with mesh-based algorithms, for example to solve partial differential equations.
- ↘ Even if you want to focus on the mesh-based case study, you should not skip the linear algebra chapter entirely, but have at least a short look on the introduction of caches and respective models to analyse cache performance (Sects. 13.1 and 13.2).
- ↰ If you skipped the chapter on locality properties, you should have at least a short look at the basics provided there, before you proceed.

Exercises

12.1. Formulate an arithmetisation of the 2D quasi-Sierpinski curve, following the approach introduced in Chap. 4.

12.2. Try to find an exponent r, for which the Hölder continuity holds for the Sierpinski curve of Sect. 8.3.

Hint: You can show that any edge is bisected, i.e. cut to half of its length, after at most five bisection steps.

12.3. Examine the refinement edges of the tetrahedra used for the construction of the Sierpinski curve of Sect. 8.3. Show that only *black* tetrahedra occur.

12.4. For the canonical tetrahedron T_{can}, as given in Eq. (12.3), perform the first three bisection steps according to Kossaczky's notation. Compute the edge lengths of the eight resulting tetrahedra to show that they are all congruent to each other.

12.5. Determine the order of vertices $x_1, x_2, x_3, x_4 \in \mathbb{R}^3$ in the notations for the Maubach and the Bänsch-Kossaczky scheme, such that the two schemes generate the same tree of tetrahedral elements.

Chapter 13
Case Study: Cache Efficient Algorithms for Matrix Operations

In Chaps. 10 and 11, we discussed applications of space-filling curves for parallelisation, which were motivated by their locality properties. In the following two chapters, we will discuss further applications, which again exploit the intrinsic locality properties of space-filling curves. As both applications will focus on inherently cache-efficient algorithms, we will start with a short introduction to cache-memory architectures, and discuss the resulting requirements on cache-efficient algorithms.

13.1 Cache Efficient Algorithms and Locality Properties

In computer architecture, a so-called *cache* (or cache memory) denotes a fast memory component that replicates a certain part of the main memory to allow faster access to the cached (replicated) data. Such caches are necessary, because standard memory hardware is nowadays much slower than the CPUs. The access latency, i.e. the time between a data request and the arrival of the first requested piece of data, is currently[1] about 60–70 ns. During that time, a fast CPU core can perform more than 100 floating point operations. This so-called "memory-gap" between CPU speed and main memory is constantly getting worse, because CPU speed is improving much faster (esp. due to using multicore processors) than memory latency. For memory bandwidth (i.e. the maximum rate of data transfer from memory to CPU), the situation is a bit better, but there are already many applications in scientific computing whose performance is limited by memory bandwidth instead of CPU speed.

Cache memory can be made much faster, and can even keep up with CPU speed, but only if it is much smaller than typical main memory. Hence, modern workstations use a hierarchy of cache memory: a very small, so-called first-level

---

[1] Currently, here, means in the year 2010.

M. Bader, *Space-Filling Curves*, Texts in Computational Science and Engineering 9,
DOI 10.1007/978-3-642-31046-1_13, © Springer-Verlag Berlin Heidelberg 2013

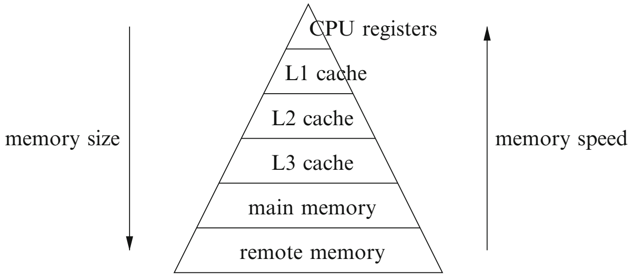

CPU registers
L1 cache
L2 cache
L3 cache
main memory
remote memory
memory size
memory speed

Fig. 13.1 A typical pyramid of hierarchical cache memory

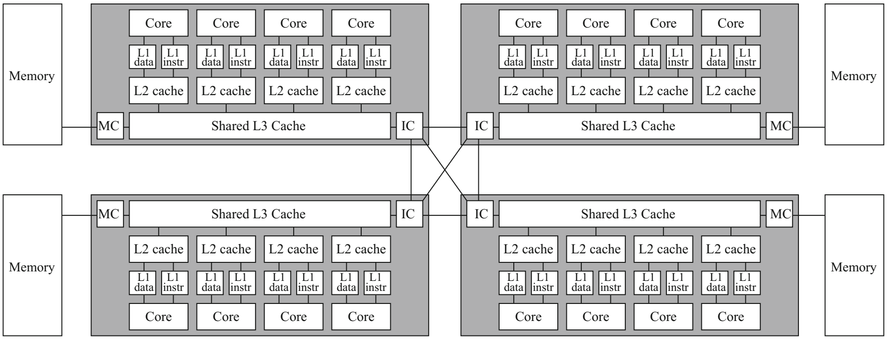

Core
L1 data
L1 instr
L2 cache
Shared L3 Cache
MC
IC
Memory

Fig. 13.2 Four quadcore processors (with shared L3 cache) with a NUMA interconnect (IC)

cache, running at the same speed as the CPU; a second-level cache that is larger (typically a few megabytes), but only running at half speed, or less; and maybe further levels that are again larger, but slower. Figure 13.1 illustrates such a pyramid of cache memory.

The situation is further complicated on multi- and manycore CPUs, because cache memory can then be shared between the CPU cores of one processor, and, in addition, we may have non-uniform memory access (NUMA) to main memory. Figure 13.2 shows a schematic diagram of four quadcore CPUs, where in each CPU, all cores share a common level-3 cache, but have individual level-1 and level-2 caches. Each CPU has a separate memory controller (MC), which is connected to a part of main memory – access to non-local memory runs via an interconnect (IC) between the four cores. As a result, CPUs will have different access speeds to different parts of the main memory.

Memory-Bound Performance and Cache Memory

The size and speed of the individual cache levels will have an interesting effect on the runtime of software. In the classical analysis of the runtime complexity of

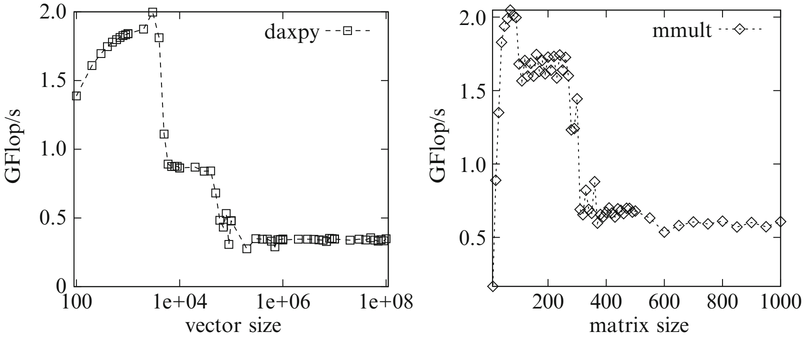


Fig. 13.3 Performance of an optimized vector operation (`daxpy`-operator: $y := \alpha x + y, \alpha \in \mathbb{R}$) and of a naive implementation of matrix multiplication for increasing problem size. Note the substantial performance drops and the different performance levels, which result from data fitting into the cache levels

algorithms, a typical job is to determine how the runtime depends on the input size, which leads to the well-known $\mathcal{O}(N)$ considerations. There, we often assume that all operations are executed with the same speed, which is no longer true on cache-based systems.

Instead, we often observe a situation as in Fig. 13.3. For very small problem size, all data resides in the first-level cache, and the algorithm runs at top speed. As soon as the data does no longer fit into level-1 cache, the speed is reduced to a level determined by the level-2 cache. The respective, step-like decrease of execution speed is repeated for every level of the memory hierarchy, and leads to the velocity profile given in Fig. 13.3. While the complexity of an algorithm will not change in the $\mathcal{O}(N)$-sense, such large differences in execution speed cannot be ignored in practice. Hence, we need to make implementations cache efficient, in order to fully exploit the available performance.

How Cache Memory Works: Cache Lines and Replacement Strategies

Once we use more data than we can fit into the cache, we need a mechanism to decide what data should be in the cache at what time. For such cache strategies, there are a couple of technical and practical restrictions:

- For technical reasons, caches do not store individual bytes or words, but small contiguous chunks of memory, so-called *cache lines*, which are always transferred to and from memory as one block. Hence, cache lines are mapped to corresponding lines in main memory. To simplify (and, thus, speed up) this mapping, lines of memory can often be transferred only to a small subset of lines: we speak of an *n-associative* cache, if a memory line can be kept in n different cache lines. The most simple case, a 1-associative cache, is called a *direct-mapped cache*; a fully associative cache, where memory lines may be kept in any cache line, are powerful, but much more expensive to build (if the cache access needs to stay fast).

- If we want to load a specific cache line from memory, but already have a full cache, we naturally have to remove one cache line from the cache. In an ideal case, we would only remove cache lines that are no longer accessed. As the cache hardware can only guess the future access pattern, certain heuristics are used to replace cache lines. Typical strategies are to remove the cache line that was not used for the longest time (*least recently used*), or which had the fewest accesses in the recent history (*least frequently used*).
- A programmer typically has almost no influence on what data is kept in the cache. Only for loading data into the cache, so-called prefetching commands are sometimes available.

Associativity, replacement strategy, and other hardware properties may, of course, vary between the different levels of caches. And it is also clear that these restrictions have an influence of the efficiency of using caches.

Cache Memory and Locality

Caches lead to a speedup, if repeatedly accessed data is kept in the cache, and is thus accessed faster. Due to the typical cache design, algorithms and implementations will be cache efficient, if their data access pattern has good *temporal* or *spatial* locality properties:

- *Temporal locality* means that a single piece of data will be repeatedly accessed during a short period of time. Replacement strategies such as *least recently used* or *least frequently used* will then reduce the probability of removing this data from the cache to a minimum.
- *Spatial locality* means that after an access to a data item, the next access(es) will be to items that are stored in a neighbouring memory address. If it belongs to the same cache line as the previously accessed item, it has been loaded into the cache as well, and can be accessed efficiently.

Hence, the cache efficiency of an algorithm depends on its temporal and spatial locality properties.

Cache-Aware and *Cache-Oblivious* Algorithms

Cache efficient implementation or algorithms can be classified into two categories, depending on whether they consider the exact cache architecture of a platform:

- *Cache-aware* algorithms or implementations use detailed information about the cache architecture. They try to increase the temporal or spatial locality by adapting the access pattern to exactly fit the number and size of the cache levels, the length of the cache line, etc. Hence, such a cache-aware implementation is specific for a particular platform, and at least certain parameters need to be tuned, if the architecture changes.

- *Cache-oblivious* algorithms, in contrast, do not use explicit knowledge on a specific architecture. Instead, the algorithms are designed to be inherently cache efficient, and profit from any presence of caches, regardless of their size and number of cache levels. Hence, their data access pattern need to have excellent temporal and spatial locality properties.

We have seen that space-filling curves are an excellent tool to create data structures with good locality properties. Hence, we will discuss some examples of how these properties can be exploited to obtain *cache-oblivious* algorithms.

13.2 Cache Oblivious Matrix-Vector Multiplication

For a vector $x = (x_1, \dots, x_n) \in \mathbb{R}^n$ and an $n \times n$-matrix A with elements a_{ij}, the elements of the matrix-vector product $y = Ax$ are given as

$$y_i := \sum_{j=1}^{n} a_{ij} x_j \,.$$

The matrix-vector product is a standard task in linear algebra, and also not difficult to implement, which makes it a popular programming task in introductory lectures on programming. We can assume that most implementations will look similar to the one given in Algorithm 13.1 (omitting the initialisation of the vector y).

Algorithm 13.1: Matrix-vector multiplication (loop-based)

```
for $i = 1 \dots n$ do
    for $j = 1 \dots n$ do
        $y_i := y_i + A_{ij} * x_j$ ;
    end
end
```

To determine the cache efficiency of this algorithm, let's now examine the temporal and spatial locality properties of this implementation. Regarding the matrix A, we notice that each element is accessed exactly once. Hence, temporal locality of the access will not be an issue, as no element will be reused. The spatial locality depends on the memory layout of the matrix. If the elements are stored in the same sequence as they are traversed by the two nested loops, the spatial locality will be optimal. Hence, for Algorithm 13.1, the matrix elements should be stored row-by-row (so-called *row-major* layout). However, if our programming language uses *column-major* layout (as in FORTRAN, for example), we should change Algorithm 13.1: the spatial locality would then be optimal, if we exchange the i- and j-loop. In general, spatial locality is perfect as long as we traverse the matrix element in the same order as they are stored in memory.

Changing the traversal scheme of the matrix elements will, however, also change the access to the two vectors x and y. As both vectors are accessed or updated n times throughout the execution of the algorithm, both temporal and spatial locality are important. For the loop-based access given by Algorithm 13.1, the access to both vectors is spatially local, even if we exchange the two loops. However, the temporal locality is different for the two vectors. The access to vector y is optimal: here, all n accesses to a single element are executed, before we move on to the next element. For the access to x, we have exactly the opposite situation: Before an element of x is re-used, we first access all other elements of the vector. Hence, the temporal locality for this pattern is the worst we can find. A first interesting question is, whether exchanging the loops will improve the situation. Then, x and y will change roles, and which option is faster depends on whether temporal locality is more important for the read access to x or for the write access to y. Even more interesting is the question whether we can derive a traversal of the matrix elements that leads to more local access patterns to both vectors.

Matrix Traversals Using Space-Filling Curves

In Algorithm 13.1, we easily see that the two loops may be exchanged. Actually, there is no restriction at all concerning the order in which we execute the element operations. We should therefore denote Algorithm 13.1 in the following form:

Algorithm 13.2: Matrix-vector multiplication (forall loop)

```
forall the $(i, j) \in \{1, \dots, n\}^2$ do
    $y_i := y_i + A_{ij} * x_j$ ;
end
```

From Algorithm 13.2, we can now consider execution orders without being influenced by loop constructions. We just need to make sure that each matrix element is processed exactly once, i.e. that we perform a *traversal*, and can then try to optimise the temporal and spatial locality properties. Our previous discussions of the locality properties of space-filling curves therefore provide us with an obvious candidate.

Assuming that the matrix dimension n is a power of two, we can use a Hilbert iteration, for example, to traverse the matrix. Hence, we modify the traversal algorithm of Chap. 3.2 to obtain an algorithm for matrix-vector multiplication. For that purpose, it is sufficient to interpret our direction operations `up`, `down`, `left`, and `right` as index operations on the matrix and the two vectors. Operators `up` and `down` will increase or decrease i, i.e., the current row index in A and vector index in y. Operators `left` and `right` will change j, which is the column index of A and the vector index of x, respectively. Algorithm 13.3 outlines the full algorithm for matrix-vector multiplication.

Algorithm 13.3: Matrix-vector multiplication based on Hilbert order matrix traversal

```
Procedure mulvH($n$)
    Parameter: $n$: matrix/vector size (must be a power of 2)
    Data: $A,x$: input matrix and vector; $i,j$: current row/column
    Result: $y$: matrix-vector product (initialised to 0 at entry)
begin
    if $n = 1$ then
        y[i] = y[i] + A[i,j] * x[j];
    else
        mulvA($n/2$); i++;
        mulvH($n/2$); j++;
        mulvH($n/2$); i—;
        mulvB($n/2$);
    end
end
...
```

To retain the optimal locality of the access to the matrix elements, the matrix elements need to be stored according to the Hilbert order, as well. For the accesses to the vectors x and y, we obtain an overall improved temporal locality. During the matrix traversal, all elements of a $2^k \times 2^k$-block will be processed before the Hilbert order moves on to the next $2^k \times 2^k$-block. During the corresponding $(2^k)^2$ element operations, 2^k elements of vector x and 2^k elements of vector y will be accessed – each of them 2^k times. On average, we will therefore execute m^2 operations on a working set of only $2m$ elements – for any $m \leq n$. Hence, even if only a small amount of elements will fit in a given cache, we can guarantee re-use of both elements of x and y. Our “naive” implementation in Algorithm 13.1 will guarantee this only for one of the two vectors.

13.3 Matrix Multiplication Using Peano Curves

For two $n \times n$-matrices A and B, the elements of the matrix product $C = AB$ are given as

$$C_{ik} := \sum_{j=1}^{n} A_{ij} B_{jk}.$$

Similar to Algorithm 13.1 for the matrix-vector product, we could implement the computation of all C_{ik} via three nested loops – one over j to compute an individual element C_{ik}, and two loops over i and k, respectively – compare Algorithm 13.4

Algorithm 13.4: Matrix multiplication (loop-based)

```
for $i = 1 \dots n$ do
    for $k = 1 \dots n$ do
        for $j = 1 \dots n$ do
            $C_{ik} := C_{ik} + A_{ij} * B_{jk}$; ;
        end
    end
end
```

Again, we can exchange the three loops arbitrarily. However, for large matrices, the cache efficiency will be less than optimal for any order. In library implementations, multiple optimisation methods, such as loop unrolling or blocking and tiling, are applied to improve the performance of matrix multiplication. The respective methods carefully change the execution order to match the cache levels – in particular the sizes of the individual caches (see the references section at the end of this chapter). In the following, we will discuss an approach based on Peano curves, which leads to a cache oblivious algorithm, instead.

As for matrix-vector multiplication, we start with Algorithm 13.5, where we stress that we simply have to execute n^3 updates $C_{ik} = C_{ik} + A_{ij} B_{jk}$ for all triples (i, j, k). Thus, matrix multiplication is again an instance of a traversal problem.

Algorithm 13.5: Matrix multiplication (forall loop)

```
forall the $(i, j, k) \in \{1, \dots, n\}^3$ do
    $C_{ik} := C_{ik} + A_{ij} * B_{jk}$ ;
end
```

The n^3 element operations correspond to a 3D traversal of the triple space. For the access to the matrix elements, in contrast, only two out of three indices are needed for each of the involved matrices. Hence, the indices are obtained via respective projections. We will therefore use a Peano curve for the traversal, because the projections of the classical 3D Peano curve to 2D will again lead to 2D Peano curves. In Fig. 13.4, this property is illustrated for the vertical direction. It holds for the other two coordinate directions, as well.

Figure 13.4 also tells us that if we execute the matrix operations $C_{ik} = C_{ik} + A_{ij} B_{jk}$ according to a 3D Peano order, the matrix elements will be accessed according to 2D Peano orders. As a consequence, we should store the elements in

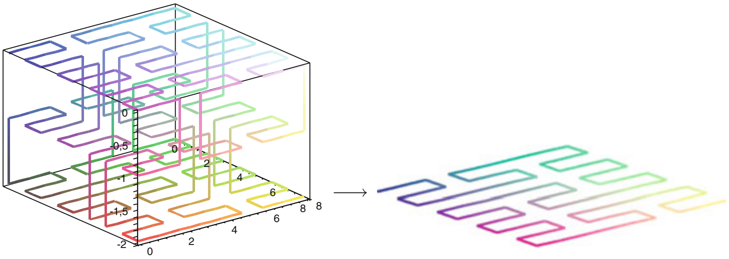


Fig. 13.4 Illustration of the projection property of the 3D Peano curve

that order, if possible. We will first try this for the simple case of a 3 × 3 matrix multiplication:

$$\underbrace{\begin{pmatrix} c_0 & c_5 & c_6 \\ c_1 & c_4 & c_7 \\ c_2 & c_3 & c_8 \end{pmatrix}}_{=:\,C} \mathrel{+}= \underbrace{\begin{pmatrix} a_0 & a_5 & a_6 \\ a_1 & a_4 & a_7 \\ a_2 & a_3 & a_8 \end{pmatrix}}_{=:\,A} \underbrace{\begin{pmatrix} b_0 & b_5 & b_6 \\ b_1 & b_4 & b_7 \\ b_2 & b_3 & b_8 \end{pmatrix}}_{=:\,B} \tag{13.1}$$

(here, the element indices indicate the 2D Peano element order). Then, the 3D Peano traversal of the element operations will lead to the following sequence of element updates:

$$\begin{tikzcd}
c_0 \mathrel{+}= a_0b_0 \arrow[d] & c_5 \mathrel{+}= a_6b_3 \arrow[r] & c_5 \mathrel{+}= a_5b_4 \arrow[d] & c_6 \mathrel{+}= a_5b_7 \arrow[r] & c_6 \mathrel{+}= a_6b_8 \arrow[d] \\
c_1 \mathrel{+}= a_1b_0 \arrow[d] & c_4 \mathrel{+}= a_7b_3 \arrow[u] & c_4 \mathrel{+}= a_4b_4 \arrow[d] & c_7 \mathrel{+}= a_4b_7 \arrow[u] & c_7 \mathrel{+}= a_7b_8 \arrow[d] \\
c_2 \mathrel{+}= a_2b_0 \arrow[d] & c_3 \mathrel{+}= a_8b_3 \arrow[u] & c_3 \mathrel{+}= a_3b_4 \arrow[d] & c_8 \mathrel{+}= a_3b_7 \arrow[u] & c_8 \mathrel{+}= a_8b_8 \\
c_2 \mathrel{+}= a_3b_1 \arrow[d] & c_2 \mathrel{+}= a_8b_2 \arrow[u] & c_3 \mathrel{+}= a_2b_5 \arrow[d] & c_8 \mathrel{+}= a_2b_6 \arrow[u] & \\
c_1 \mathrel{+}= a_4b_1 \arrow[d] & c_1 \mathrel{+}= a_7b_2 \arrow[u] & c_4 \mathrel{+}= a_1b_5 \arrow[d] & c_7 \mathrel{+}= a_1b_6 \arrow[u] & \\
c_0 \mathrel{+}= a_5b_1 \arrow[r] & c_0 \mathrel{+}= a_6b_2 \arrow[u] & c_5 \mathrel{+}= a_0b_5 \arrow[r] & c_6 \mathrel{+}= a_0b_6 \arrow[u] &
\end{tikzcd} \tag{13.2}$$

Note that this scheme follows an inherently local access pattern to the elements: after each element operation, the next operation will re-use one element and access two elements that are direct neighbours of the elements accessed in the previous operation. To extend this simple 3 × 3-scheme to a multiplication algorithm for larger matrices, we need

- A 2D Peano order that defines the data structure for the matrix elements;
- A recursive extension of the 3×3-scheme in Eq. 13.2, which is basically obtained by using matrix blocks instead of elements;

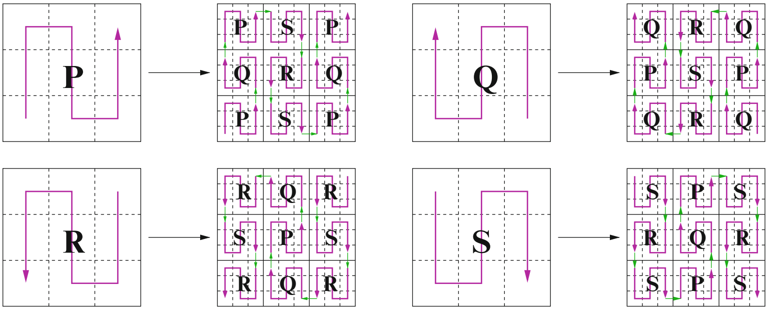


Fig. 13.5 Recursive construction of the Peano element order to store matrices

- A concept for matrices of arbitrary size, as the standard Peano order will only work for matrices of size $3^k \times 3^k$.

The 2D Peano order for the elements is derived in a straightforward manner from the iterations of the Peano curve. The respective construction is illustrated in Fig. 13.5. The pattern symbols P, Q, R, and S now denote a numbering scheme for the corresponding subblock in the matrix.

13.3.1 Block-Recursive Peano Matrix Multiplication

Let's now formulate the 3 × 3 multiplication scheme of Eq. (13.2) to a scheme for matrices in Peano order. In a first step, we write the matrices as 3×3 block matrices:

$$\begin{pmatrix} P_{A0} & R_{A5} & P_{A6} \\ Q_{A1} & S_{A4} & Q_{A7} \\ P_{A2} & R_{A3} & P_{A8} \end{pmatrix} \begin{pmatrix} P_{B0} & R_{B5} & P_{B6} \\ Q_{B1} & S_{B4} & Q_{B7} \\ P_{B2} & R_{B3} & P_{B8} \end{pmatrix} = \begin{pmatrix} P_{C0} & R_{C5} & P_{C6} \\ Q_{C1} & S_{C4} & Q_{C7} \\ P_{C2} & R_{C3} & P_{C8} \end{pmatrix}. \tag{13.3}$$

Here, we named each matrix block according to its numbering scheme (see Fig. 13.5), and indicated the name of the global matrix and the relative position of the block in the Peano order as indices. The element operations of Eq. 13.2 now lead to multiplication of matrix blocks, as in $P_{C0} \mathrel{+}= P_{A0}P_{B0}$, $Q_{C1} \mathrel{+}= Q_{A1}P_{B0}$, $P_{C2} \mathrel{+}= P_{A2}P_{B0}$, etc.

If we just examine the numbering patterns of the matrix blocks, we see that there are exactly eight different types of block multiplications:

$$\begin{array}{llll} P \mathrel{+}= PP & Q \mathrel{+}= QP & R \mathrel{+}= PR & S \mathrel{+}= QR \\ P \mathrel{+}= RQ & Q \mathrel{+}= SQ & R \mathrel{+}= RS & S \mathrel{+}= SS. \end{array} \tag{13.4}$$

For the block operation $P \mathrel{+}= PP$, we have already derived the necessary block operations and their optimal sequence of execution in Eq. (13.2). For the other seven

Table 13.1 Execution orders for the eight different block multiplication schemes. '+' indicates that the access pattern for the respective matrix A, B, or C is executed in forward direction (from element 0 to 8). '−' indicates backward direction (starting with element 8)

Block scheme	$P \mathrel{+}= PP$	$P \mathrel{+}= RQ$	$Q \mathrel{+}= QP$	$Q \mathrel{+}= SQ$	$R \mathrel{+}= PR$	$R \mathrel{+}= RS$	$S \mathrel{+}= QR$	$S \mathrel{+}= SS$
A	+	+	+	+	−	−	−	−
Access to B	+	+	−	−	+	+	−	−
C	+	−	+	−	+	−	+	−

types of block multiplications, it turns out that we obtain similar execution patterns, and that no further block operations arise besides those already given in Eq. (13.4). Hence, we obtain a closed system of eight recursive multiplication schemes.

13.3.2 Memory Access Patterns During the Peano Matrix Multiplication

Our next task is to derive execution sequences for the other seven multiplication schemes: $Q \mathrel{+}= QP$, $R \mathrel{+}= PR$, etc. We will leave the details for Exercise 13.2, and just state that each multiplication scheme leads to an execution order similar to that given in Eq. (13.2). In addition, all eight execution orders follow the same structure as the scheme for $P \mathrel{+}= PP$, except that for one, two, or even all three of the involved matrices, the access pattern runs backwards. Table 13.1 lists for the eight different schemes which of the access patterns run backwards.

All eight schemes can be implemented by using only increments and decrements by 1 on the Peano-ordered element indices. Hence, jumps in memory are completely avoided throughout the computation of a 3 × 3 block. Our next step is therefore to make sure that jumps are also avoided between consecutive block schemes. As example, we examine the transfer between the first two block operations in a $P \mathrel{+}= PP$ scheme: we assume that we just finished the operation $P_{C0} \mathrel{+}= P_{A0} P_{B0}$, and now would go on with $Q_{C1} \mathrel{+}= Q_{A1} P_{B0}$.

- On matrix A, we have traversed the P-ordered block 0, which means that our last access was to the last element in this block. The $Q \mathrel{+}= QP$ scheme runs in '+' direction on A (see Table 13.1), such that the first access to the Q-ordered block 1 in A will be to the first element, which is of course a direct neighbour of the last element of block 0.
- In matrix C, we have the identical situation: the P-ordered block 0 has been traversed in '+' direction, and the next access will be to the first element in the Q-ordered block 1.
- In matrix B, both execution sequences work on the P-ordered block 0. However, while the $P \mathrel{+}= PP$ scheme accesses B in '+' direction, the $Q \mathrel{+}= QP$ scheme will run in '−' direction on B. Hence, the last access of the $P \mathrel{+}= PP$ scheme is to the last element of the block, which will also be the start of the $Q \mathrel{+}= QP$ scheme.

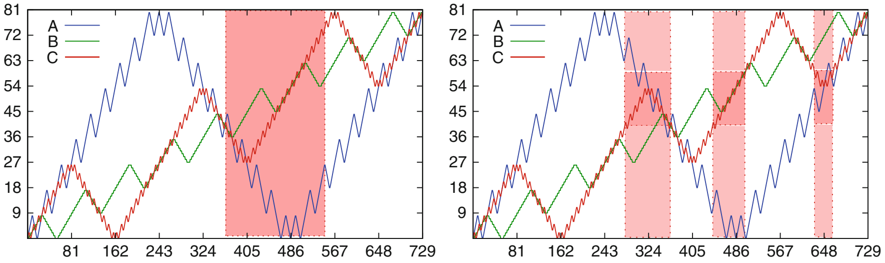


Fig. 13.6 Memory access pattern for all three matrices during a 9×9 Peano matrix multiplication. The highlighted areas are obtained by partitioning the operations (*left image*) or the accessed elements of C (*right image*)

Hence, the increment/decrement property stays valid between the two block operations. A careful analysis (which is also too tedious to be discussed here) reveals that this is true for all recursive block operations occurring in our multiplication scheme.

As a result, the Peano matrix multiplication can be implemented as illustrated in Algorithm 13.6. There, the schemes $P \mathrel{+}= PP$, $Q \mathrel{+}= QP$, etc., are coded via the '+'- or '−'-directions of the execution orders for the three matrices, as listed in Table 13.1. The change of direction throughout the recursive calls is coded in the parameters `phsA`, `phsB`, and `phsC`.

The increment/decrement property leads to an inherently local memory access pattern during the Peano matrix multiplication, which is illustrated in Fig. 13.6. There, we can observe certain locality properties of the memory access:

- A given range of contiguous operations (highlighted in the left image) will access only a certain contiguous subset of matrix elements.
- Vice versa, a contiguous subset of matrix elements will be accessed by a set of operations that consists of only a few contiguous sequences.

Both, the operator subsets and the range of elements, define partitions that can be used to parallelise the Peano matrix multiplication (following a work-oriented or an owner-computes partitioning, respectively). However, the underlying locality properties also make the algorithm inherently cache efficient, which we will examine in the following section.

13.3.3 Locality Properties and Cache Efficiency

Figure 13.6 illustrates the spatial locality properties of the new matrix multiplication. From the respective chart, we can infer a so-called *access locality function*, $L_M(n)$, which we can define as the maximal possible distance (in memory) between two elements of a matrix M that are accessed within n contiguous operations.

For a loop-based implementation of matrix multiplication, we will typically access a range of k successive elements during k operations – which already

Algorithm 13.6: Recursive implementation of the Peano matrix multiplication

```
Procedure peanoMult (phsA, phsB, phsC, dim)
    Data: A,B,C: the matrices, C will hold the result of C += AB;
          a,b,c: indices of the matrix element of A, B, and C
          (a, b, and c should be initialised to 0 on entry)
    Parameter: phsA,phsB,phsC: index increments/decrements for a, b, and c;
               dim: matrix block dimension (must be a power of 3)
begin
    if dim = 1 then
        C[c] += A[a] * B[b];
    else
        peanoMult ( phsA, phsB, phsC, dim/3) ; a += phsA; c += phsC;
        peanoMult ( phsA,–phsB, phsC, dim/3) ; a += phsA; c += phsC;
        peanoMult ( phsA, phsB, phsC, dim/3) ; a += phsA; b += phsB;

        peanoMult ( phsA, phsB,–phsC, dim/3) ; a += phsA; c –= phsC;
        peanoMult ( phsA,–phsB,–phsC, dim/3) ; a += phsA; c –= phsC;
        peanoMult ( phsA, phsB,–phsC, dim/3) ; a += phsA; b += phsB;

        peanoMult ( phsA, phsB, phsC, dim/3) ; a += phsA; c += phsC;
        peanoMult ( phsA,–phsB, phsC, dim/3) ; a += phsA; c += phsC;
        peanoMult ( phsA, phsB, phsC, dim/3) ; b += phsB; c += phsC;

        peanoMult (–phsA, phsB, phsC, dim/3) ; a –= phsA; c += phsC;
        peanoMult (–phsA,–phsB, phsC, dim/3) ; a –= phsA; c += phsC;
        peanoMult (–phsA, phsB, phsC, dim/3) ; a –= phsA; b += phsB;

        peanoMult (–phsA, phsB,–phsC, dim/3) ; a –= phsA; c –= phsC;
        peanoMult (–phsA,–phsB,–phsC, dim/3) ; a –= phsA; c –= phsC;
        peanoMult (–phsA, phsB,–phsC, dim/3) ; a –= phsA; b += phsB;

        peanoMult (–phsA, phsB, phsC, dim/3) ; a –= phsA; c += phsC;
        peanoMult (–phsA,–phsB, phsC, dim/3) ; a –= phsA; c += phsC;
        peanoMult (–phsA, phsB, phsC, dim/3) ; b += phsB; c += phsC;

        peanoMult ( phsA, phsB, phsC, dim/3) ; a += phsA; c += phsC;
        peanoMult ( phsA,–phsB, phsC, dim/3) ; a += phsA; c += phsC;
        peanoMult ( phsA, phsB, phsC, dim/3) ; a += phsA; b += phsB;

        peanoMult ( phsA, phsB,–phsC, dim/3) ; a += phsA; c –= phsC;
        peanoMult ( phsA,–phsB,–phsC, dim/3) ; a += phsA; c –= phsC;
        peanoMult ( phsA, phsB,–phsC, dim/3) ; a += phsA; b += phsB;

        peanoMult ( phsA, phsB, phsC, dim/3) ; a += phsA; c += phsC;
        peanoMult ( phsA,–phsB, phsC, dim/3) ; a += phsA; c += phsC;
        peanoMult ( phsA, phsB, phsC, dim/3) ;
    end
end
```

requires some optimisations in order to avoid stride-n accesses. For the naive implementation given in Algorithm 13.4, we thus have $L_M(n) \geq n$ for all involved matrices. For an improved algorithm that operates on matrix blocks of size $k \times k$, we will typically obtain loops over k contiguous elements, so the worst case will reduce to $L_M(k) \geq k$ with $k \ll n$. However, as long as we stay with a $k \times k$ block, we will perform k^3 operations on only k^2 elements of a matrix. Hence, if our storage scheme for the matrices uses $k \times k$-blocks, as well, and stores such elements contiguously, we perform k^3 operations on k^2 contiguous elements. The best case that could be achieved for the access locality function L_M should therefore be $L_A(k) \approx k^{2/3}$. Thus, even a blocked approach has the following limitations on L_M:

- We have only one block size, k_0, that can lead to the optimal locality – this could be cured, if we change to a recursive blocking scheme (a first step towards space-filling curves).
- The locality will still be $L_M(k) \geq k$, if we are within the smallest block.
- Between two blocks, the locality will only be obtained, if two successively accessed blocks are stored contiguously in memory.

The recursive blocking and the last property (contiguity) is exactly what is achieved by the Peano multiplication scheme. We can therefore obtain $L(k) \in \mathcal{O}(k^{2/3})$ as an upper bound of the extent of the index range for any value of k.

While we have $L(k) = k^{2/3}$, if we exactly hit the block boundaries, i.e., compute the distances for the first element operations of two consecutive block multiplications, we obtain a slightly worse ratio for arbitrary element operations. For a tight estimate, we need to determine the longest streak of not reusing matrix blocks in the Peano algorithm. For matrix B, which is always reused three times, the longest streak is two such block multiplications. For three block multiplications, either the first two or the last two work on the same B-block. In the scheme for matrix A, up to nine consecutive block multiplications can occur until a block is re-used. For matrix C, three contiguous block operations occur. During recursion, though, two such streaks can occur right after each other. For matrix A, we therefore obtain 18 as the length of the longest streak. In the worst case, we can therefore do $18n^3$ block operations on matrix A on $18n^2$ contiguous elements of A. Thus, for matrix A, we get that

$$L_A(n) \approx \frac{18}{18^{2/3}} n^{2/3} = \sqrt[3]{18} n^{2/3}. \tag{13.5}$$

For matrix C, we get a maximum streak length of 6, and therefore

$$L_B(n) \approx \sqrt[3]{2} n^{2/3}, \qquad L_C(n) \approx \sqrt[3]{6} n^{2/3}. \tag{13.6}$$

If we only consider $\mathcal{O}(n^3)$-algorithms for matrix multiplication, i.e., if we disregard Strassen's algorithm and similar approaches, then a locality of $L_B(n) \in \mathcal{O}(n^{2/3})$ is the optimum we can achieve. The locality functions $L_A(n) \leq 3n^{2/3}$, $L_B(n) \leq 2n^{2/3}$, and $L_C(n) \leq 2n^{2/3}$ are therefore asymptotically optimal. L_B and L_C, in addition, involve very low constants.

13.3.4 Cache Misses on an Ideal Cache

The access locality functions provide us with an estimate on how many operations will be performed on a given set of matrix elements. If a set of elements is given by a cache line or even the entire cache, we can estimate the number of cache hits and cache misses of the algorithm. To simplify that calculation, we use the model of an *ideal cache*, which obeys to the following assumptions:

- The cache consists of M words that are organized as cache lines of L words each. Hence, we have M/L cache lines. The external memory can be of arbitrary size and is structured into memory lines of L words.
- The cache is fully associative, i.e., can load any memory line into any cache line.
- If lines need to be evicted from the cache, we assume that the cache can "foresee the future", and will always evict a line that is no longer needed, or accessed farthest in the future.

Assume that we are about to start a $k \times k$ block multiplication, where k is the largest power of 3, such that three $k \times k$ matrices fit into the cache. Hence, $3 \cdot k^2 < M$, but $3 \cdot (3k)^2 > M$, or

$$\frac{1}{3}\sqrt{\frac{M}{3}} < k < \sqrt{\frac{M}{3}}. \qquad (13.7)$$

The cache will usually (except at start) be occupied by blocks required from previous multiplications; however, one of the three involved blocks will be reused and thus already be stored in the cache. To perform the following element operations, we will successively have to fetch all cache lines that contain the elements of the two new matrix blocks. The ideal cache strategy will ensure that the matrix block that is reused from the previous block multiplication will not be evicted from the cache. Instead, cache lines from the other two blocks will be replaced by the elements for the new block operation. We can also be sure that elements of these new blocks will not be evicted during the entire block operation. Hence, there will be only $2\left\lceil \frac{k^2}{L} \right\rceil$ cache line transfers for these two $k \times k$ blocks – to simplify the following computation, we assume that $\left\lceil \frac{k^2}{L} \right\rceil = \frac{k^2}{L}$.

As we will perform $(n/k)^3$ such block operations, the total number of cache line transfers throughout an $n \times n$ multiplication will be

$$T(n) = \left(\frac{n}{k}\right)^3 \cdot 2 \cdot \left(\frac{k^2}{L}\right) = \frac{2n}{kL} \leq \left(\frac{2n}{\frac{1}{3}\sqrt{\frac{M}{3}} \cdot L}\right) = 6\sqrt{3}\frac{n^3}{L\sqrt{M}}. \qquad (13.8)$$

For arbitrary size of the cache lines, we still have $T(N) \in \mathcal{O}\left(\frac{n^3}{L\sqrt{M}}\right)$, with a constant close to $6\sqrt{3}$. Note that the respective calculation also works, if we have multiple levels of cache memory. Then, the estimate of cache line transfers refers to the respective size M of the different caches. For realistic caches, we might obtain more cache transfers because of a bad replacement strategy. However, a cache that always evicts the cache line that was used longest ago ("least recently used" strategy) will expel those cache lines first that contain matrix elements that are farthest away in terms of location in memory, because the access pattern of the multiplication will make sure that all matrix elements that are "closer" in memory have been accessed at a later time. What we cannot consider at this point is effects due to limited associativity of the cache, i.e., if a cache can place memory lines only into a specific set of cache lines.

13.3.5 Multiplying Matrices of Arbitrary Size

For practical implementations, it turns out that regular CPUs will not run at full performance, if we use a fully recursive implementation. For small matrix blocks, loop-based implementations are much more efficient. One of the reasons is that vector-computing extensions of CPUs can then be exploited. We should therefore stop the recursion on a certain block size $k \times k$, which is chosen to respect such hardware properties. To extend our algorithm for matrices of arbitrary size, we then have three options:

1. We can embed the given matrices into larger matrices of size $3^p \times 3^p$ (or, with blocks as leaves: $3^p k \times 3^p k$). The additional zeros should, of course, not be stored. A respective approach is described in Sect. 13.4, where such a *padding* approach is given for band matrices or sparse matrices in general. In such an approach, we will typically stop the recursion on matrix blocks, as soon as these fit into the innermost cache of a CPU.
2. In Sect. 8.2.3, we introduced Peano iterations on 3D grids of size $k \times l \times m$, where k, l, and m may be arbitrary odd numbers. The Peano matrix multiplication also works on such Peano iterations. For the leaf-block operations, we have to introduce schemes for matrix multiplication on $n_1 \times n_2$, where n_1 and n_2 may be 3, 5, or 7, respectively. Actually, the scheme will work for leaf blocks of any odd size.
3. We can stick to the classical 3×3 recursion, if we stop the recursion on larger block sizes. If these are stored in regular row- or column-major order (compare the approach in Sect. 13.4), we just need to specify an upper limit for the size of these blocks, and can then use an implementation that is optimised for small (but arbitrary) matrix sizes.

What approach performs best will depend on the specific scenario. The first approach is interesting, for example, if we can choose the size of the leaf-level matrix block such that three such blocks fit exactly into the innermost cache. The constant factor for the number of cache line transfers, as estimated in Eq. (13.8) will then be further reduced. The third approach is especially interesting for coarse-grain parallel computations. There, the leaf-level blocks will be implemented by a call to a sequential library, where the size of the leaf blocks is of reduced influence. See the works indicated in the references section for more details.

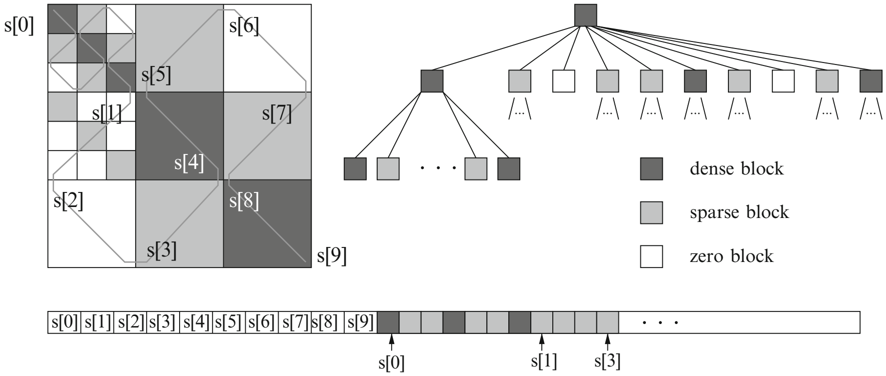


Fig. 13.7 Illustration of a spacetree storage scheme for sparse matrices. The tree representation of the sparsity pattern is sequentialised according to modified depth-first scheme, where information on the child tree is stored within a parent node

13.4 Sparse Matrices and Space-Filling Curves

Sparse matrices are matrices that contain so many zero elements that it becomes worthwhile to change to different data structures and algorithms to store and process them.[2] Considering our previous experience with space-filling-curve data structures for matrices and arrays, but also for adaptively refined grids, we can try to use a quadtree-type structure to store matrices that contain a lot of zeros. As illustrated in Fig. 13.7, we recursively subdivide a matrix into smaller and smaller blocks. The substructuring scheme can be a quadtree scheme, but due to our previously introduced Peano algorithm, we again use a 3×3 refinement, i.e., a 3^2-spacetree. Once a matrix block consists entirely of zero elements, we can stop refinement, mark the respective tree with a zero-block leaf, and thus not store the individual elements. For blocks that contain elements, we stop the recursion on blocks of a certain minimal sizes. On such blocks, we either store a dense matrix (could be even in row- or column-major order) or a small sparse matrix (using a respective simple storage scheme). The tree structure for such a storage scheme is also illustrated in Fig. 13.7.

The matrix blocks, either sparse blocks or dense blocks, are stored in the sequential order given by the Peano curve. However, in contrast to the Peano scheme for dense blocks, the matrix blocks will now have varying size – because of the different sparsity patterns of the individual blocks, but also because the zero blocks are not stored at all. Hence, to access a specified block, it will usually be necessary to traverse the sparsity tree from its root. To issue recursive calls on one or several child matrix blocks, a parent node needs information on the exact position of all child blocks in the data structure. Thus, in a node, we will store the start address of all nine child blocks of the matrix. Zero blocks are automatically considered by this scheme by just storing two consecutive identical start addresses. As we typically need both start and end addresses for the children, we also need to store the end address of the last block. Hence, for every parent node, we will store a sequence of ten integers, as indicated by the data stream in Fig. 13.7. We thus obtain a data structure that follows the same idea as the modified depth-first traversal that was introduced for the refinement trees of adaptive grids, in Sect. 10.5.

---

[2] Following a definition given by Wilkinson.

The sparsity structure information and the matrix elements of the leaf blocks can either be stored in a single data stream or in two separate streams. If we choose to store the elements together with the structure, we will only need to change the data structure for the leaf blocks. These blocks will then require information on the block size and type (dense or sparse), as well as on the extent of the block in terms of bytes in memory. If matrix elements are stored in a separate stream, the pre-leaf-level nodes need to store respective pointers to the start of the respective blocks in the elements stream.

As already indicated in Sect. 13.3.5, we can also use the presented sparse-matrix data structure for matrices that are only dense (or comparably dense) in certain parts of the matrix – one simple example would be band matrices. In that case, we can further simplify the data structure by allowing only dense matrix blocks in the leaves.

References and Further Readings

The Peano-curve algorithm for matrix multiplication was introduced in [24], where we particularly discussed the cache properties of the algorithm. The locality properties, as given in Eqs. 13.5 and 13.8 are asymptotically optimal – respective lower bounds were proven by Hong and Kung [132]. Hardware-oriented, efficient implementations of the algorithms, including the parallelisation on shared-memory multicore platforms, were presented in [21, 127]. In [19], we discussed the parallelisation of the algorithm using message passing on distributed-memory. There, the result for cache line transfers in the ideal-cache model can be used to estimate the number of transferred matrix blocks in a distributed-memory setting. The extension for sparse matrices was presented in [22], which also includes a discussion of LU-decomposition based on the Peano matrix multiplication.

Since the advent of cache-based architectures, improving the cache efficiency of linear algebra operations has been an active area of research. Blocking approaches to improve the cache-efficiency were already applied in the first level-3 BLAS routines, when introduced by Dongarra et al. [78]. Blocking approaches and the block-oriented matrix operations were also a driving factor in the development of LAPACK [13], whose implementation was consequently based on exploiting the higher performance of BLAS 3 routines. Since then, blocking and tiling of matrix operations has become a standard technique for high performance libraries. The ATLAS project [267] uses automatic tuning of blocking sizes to the available memory hierarchy, and GotoBLAS [104] explicitly considers the *translation look-aside buffer* (TLB) and even virtual memory as further cache levels [103].

Block matrix layouts to improve cache efficiency were introduced by Chatterjee et al. [65], who studied Morton order and 4D tiled arrays (i.e., 2D arrays of matrix blocks), and by Gustavson [116], who demonstrated that recursive blocking automatically leads to cache-efficient algorithms. Frens and Wise [90] used quadtree decompositions of dense matrices and respective recursive implementations of matrix multiplication and of QR decomposition [91]. The term *cache-oblivious* for such inherently cache-efficient algorithms has been introduced by Frigo et al. [92]. For a review of both cache-oblivious and cache-aware algorithms in linear algebra, see Elmroth et al. [81]. For further, recent work, see [106, 255], e.g. Yotov et al. [112,281] compared the performance of cache-oblivious algorithms with carefully tuned cache-aware approaches, and identified efficient prefetching of matrix blocks as a crucial question for recursive algorithms. For the Peano algorithm, the increment/decrement access to blocks apparently solves this problem. In any case, block-oriented data structures and algorithms are more and more considered a necessity in the design of linear algebra routines. The designated LAPACK-successor PLASMA [56, 57], for example, aims for a stronger block orientation, and similar considerations drive the FLAME project [113]. As libraries will often depend on row-major or column-major storage, changing the matrix storage to blocked layouts on-the-fly becomes necessary; respective in-place format conversions were studied in [115].

For sparse matrices, quadtree data structures were, for example, examined by Wise et al. [271, 272]. Hybrid "hypermatrices" that mix dense and sparse blocks in recursively structured matrices were discussed by Herrero et al. [130]. Haase [117] used a Hilbert-order data structure for sparse-matrix-vector multiplication to improve cache efficiency.

Chapter 14 will present a cache-oblivious approach to solve partial differential equations on adaptive discretisation grids – hence, we save all further references to work on cache oblivious algorithms, be it grid-based or other simulation approaches, for the references section of Chap. 14.

What's next?

- The next chapter will do a second case study, now focusing on mesh-based algorithms, for example to solve partial differential equations.
- The next chapter will use Sierpinski curves and also the Lebesgue curve and Morton order – if you skipped the respective chapters and sections, you should do them now (Chaps. 6 and 12 for Sierpinski curves, Sect. 7.2 for Lebesgue curves and Morton order).

Exercises

13.1. Try to determine the number of cache misses caused by Algorithm 13.3, following the same cache model as in Sect. 13.3.4. Focus, in particular, on the misses caused by the access to the vectors x and y.

13.2. Use the graph-based illustration of Eq. (13.2) to derive the execution orders for the matrix multiplication schemes $Q \mathrel{+}= QP$, $R \mathrel{+}= PR$, $S \mathrel{+}= QR$, etc.

13.3. Consider an algorithm that uses 2D Morton order to store the matrix elements, and 3D Morton order to execute the individual element operations of matrix multiplication. Analyse the number of cache misses causes by this algorithm, using a cache model and computation as in Sect. 13.3.4.

13.4. As a data structure for sparse matrices, we could also refine the sparsity tree up to single elements, i.e., not stop the recursion on larger blocks. Give an estimate on how much memory we will require to store a sparse matrix (make suitable assumptions on the sparsity pattern or sparsity tree, if necessary).

Chapter 14
Case Study: Numerical Simulation on Spacetree Grids Using Space-Filling Curves

In Chap. 1, we have already introduced the numerical solution of a heat equation on adaptive grids as a motivating example. In Chaps. 9 and 10, we discussed how to combine spacetrees and space-filling curves to obtain efficient data structures, as well as partitioning and load distribution algorithms for the parallelisation of such problems. Together, spacetrees and space-filling curves were shown to be able to allow adaptive refinement of meshes, but still retain locality properties on the discretisation mesh. However, for full-featured numerical simulations, we still have to discuss how to deal with the following problems:

- Adaptive refinement and coarsening of the grid might require certain balancing operations on the grid. For example, certain limitations between the size of neighbouring grid cells are sometimes prescribed. Hence, we require respective balancing algorithms.
- To solve numerical problems, we need efficient algorithms to perform the subsequent computational steps of classical iterative solvers. Hence, we need efficient traversal algorithms to update the unknowns on a spacetree grid.

Additional challenges might arise from the integration of multigrid solvers, or other efficient numerical approaches.

In the following, we will extend the concept of locality properties and introduce inherently memory-local and memory-efficient traversal algorithms on spacetrees, which can be used to implement conjugate gradient or multigrid solvers for stationary problems, or time-stepping schemes to solve time-dependent equations.

14.1 Cache-Oblivious Algorithms for Element-Oriented Traversals

We will start with the heat equation problem we already used in the introduction:

- There, we used a uniformly refined discretisation grid, where the unknowns (temperature values) are placed on the vertices of the grid cells.

M. Bader, *Space-Filling Curves*, Texts in Computational Science and Engineering 9,
DOI 10.1007/978-3-642-31046-1_14, © Springer-Verlag Berlin Heidelberg 2013

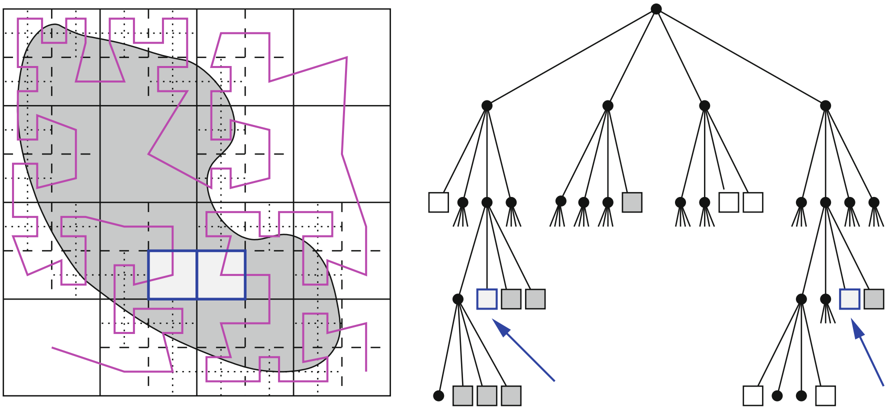


Fig. 14.1 Neighbouring unknowns in a quadtree grid

- We need to solve a system of linear equations, where the individual equations couple unknowns on adjacent grid points, for example

$$u_{i,j} - \frac{1}{4}\left(u_{i-1,j} + u_{i,j-1} + u_{i+1,j} + u_{i,j+1}\right) = f_{i,j} \qquad \text{for all } i, j \tag{14.1}$$

  in the case of a regular grid – see also Eq. (1.1).
- We will use iterative solvers to compute approximate solutions for the unknowns $u_{i,j}$. A lot of these solvers will be based on computing the residuals

$$r_{i,j} = f_{i,j} - u_{i,j} + \frac{1}{4}\left(u_{i-1,j} + u_{i,j-1} + u_{i+1,j} + u_{i,j+1}\right) \qquad \text{for all } i, j, \tag{14.2}$$

  and use the residuals for local updates of the approximate solutions.

A crucial component of our data structures and algorithms will therefore be to ensure an efficient access to the local neighbours of an unknown. In a spacetree grid, this is easy as long as a neighbouring unknown is located in the same spacetree cell. However, as Fig. 14.1 shows, neighbouring unknowns can also be located in entirely different subtrees. There are not too many choices to implement such accesses efficiently:

- If the tree structure is stored explicitly, we can follow the parent–child pointers. The number of pointers depends on the depth of the tree, and will therefore depend logarithmically on the number of grid cells. Usually, this is too expensive.
- A sequentialised storage scheme, as introduced in Sect. 9.2, will be even worse, as we would need to traverse all intermediate subtrees, which leads to computational effort which, in the worst case, is proportional to the entire number of grid cells.

- For a sequentially stored uniform grid, we can compute the position of an unknown directly, using the indexing algorithms for space-filling curves. The effort then depends on the number of digits we need to compute to determine the index, which again depends logarithmically on the grid size. For adaptive grids, however, the index computation becomes complicated, because we do not know the size of the subtrees. A modified depth-first storage, as introduced in Sects. 10.5 and 13.4, is then required, which leads to similar computational costs as a pointer-based scheme.
- To store indices of the neighbours or pointers to them explicitly will extend the tree data structure towards a general (directed, acyclic) graph. Accesses to neighbours will then have an $\mathcal{O}(1)$ computational effort, but the extension of the data structure will substantially increase the memory requirement. Depending on how much additional information we have to or choose to store, the memory advantage in comparison to using unstructured meshes will be lost.

A possible remedy is to use hash tables to store unknowns, which leads to an average complexity of $\mathcal{O}(1)$ for accesses to unknowns. However, evaluating a hash function for each access remains expensive, and hash tables are usually cache-inefficient.

The approach that we propose in the following will tackle the problem in two steps: it will avoid node-oriented traversals and use an element-oriented approach, instead, and it will introduce a stack-system to handle multiple updates of unknowns during the grid traversals.

14.1.1 Element-Based Traversals on Spacetree Grids

First of all, we note that not all neighbours are difficult to access in a spacetree: the access to unknowns within the same spacetree element should, in fact, be simple – if we disregard that unknowns on nodes and edges will belong to multiple adjacent cells. We can therefore try to traverse our grid element-by-element, only accessing the element-local unknowns in each element. As a consequence, we need to re-formulate the node-oriented computation, as suggested by the notation used in Eq. (14.1), in an element-oriented way (and, similarly, the computation of residuals, as in Eq. (14.2).

Element-Based Discretisation

Figure 14.2 illustrates how to change the node-based discretisation given by the system (14.1) into an element-based discretisation. In that way, we change a node-based system of equations $Ax = b$ into a set of element systems $A^{(e)}x^{(e)} = b^{(e)}$, where the global matrix A and right-hand side b is summed up from the element contributions:

$$A = \sum_{e \in S} A^{(e)} \qquad \text{and} \qquad b = \sum_{e \in S} b^{(e)}. \tag{14.3}$$

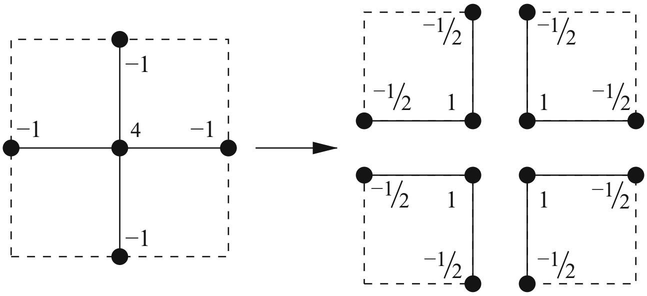


Fig. 14.2 Node-based and equivalent element-based discretisation of the heat equation

Element-oriented discretisation methods, such as Finite Element, Finite Volume, or discontinuous Galerkin methods, will directly deliver such element systems. Often, these element systems are then accumulated to a global system according to Eq. (14.3). While the following approach could also be used to assemble such global matrices, its main goal is to avoid this overhead and directly work on the element systems, instead.

Element-Based Residual Updates

Of course, we need to be able to perform operations such as computing the residual, as in Eq. (14.2), based on the element-oriented formulation. With $r = b - Ax$ and Eq. (14.3), we obtain

$$r = \sum_{e \in S} \underbrace{(b^{(e)} - A^{(e)} x^{(e)}}_{=: r^{(e)}} = \sum_{e \in S} r^{(e)}, \tag{14.4}$$

i.e. we accumulate element contributions of the global residual vector. Hence, in each element, we will access all unknowns that are located in this element, which means that all unknowns are accessed by all adjacent elements. Similarly, each component of the global residual vector corresponds to one unknown and is therefore updated by all elements adjacent to that unknown. Hence, in a rectangular grid (with unknowns located on vertices), we will have four accesses to each unknown, and also four updates to each residual.

We therefore need to work out solutions to the following two access problems:

- We need efficient access to the unknowns $x^{(e)}$ in each element. Even if unknowns are adjacent to more than one element, we only want to store them once. However, all adjacent elements need to be able to access it efficiently.
- We will do several updates on each residual component. Hence, we need to store intermediate results. As for the $x^{(e)}$, we do not want to store element contributions separately, but use only one variable to accumulate residual contributions for one unknown.

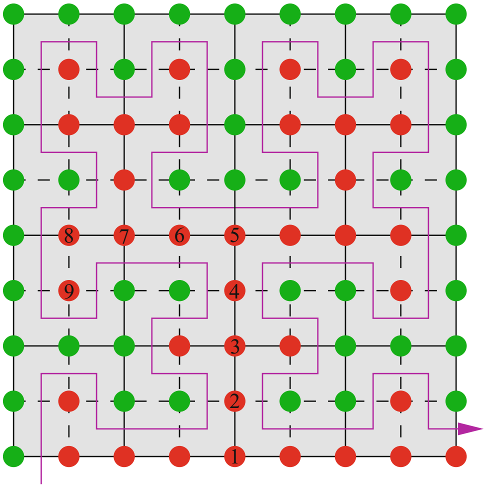


Fig. 14.3 Colouring of unknowns in a uniformly refined quadtree grid – the Hilbert curve divides the unknowns into a *green* group (*left* of the curve) and a *red* group (*right* of the curve). Note the stack principle during the access to the unknowns 1 up to 9

To put it short, we need to arrange repeated accesses and updates to given unknowns or residual components located on vertices (or edges). We do not want to store the respective variables multiple times and cannot store the variable in a fixed memory location (this would imply an index or hash-table solution, as discussed earlier). A solution to this dilemma is to store the variables only once, but in multiple locations in memory – as explained in the following paragraph.

14.1.2 Towards Stack-Based Traversals

Let's study Fig. 14.3, which shows a uniformly refined quadtree grid sequentialised by a Hilbert curve. As we have already discussed for parallelisation, in Sect. 10.6, the Hilbert iterations splits the unknowns into two halves: a *green* half that is located left of the curve, and a *red* half right of the curve. In addition the subsequent accesses to the unknowns of one colour strictly follow a *last-in-first-out* scheme, i.e. follow a stack principle. The unknowns 1–9, for example, will first be accessed in ascending order by the elements left of them. Afterwards, they will be accessed by the elements adjacent to the right-hand side in descending order. Hence, to store intermediate values of these unknowns, we can use a *stack*. To store and retrieve all unknowns of the grid, we require two different stacks – one for the *green* unknowns and one for the *red* ones. After leaving an element, we will store all unknowns that need to be accessed again on the corresponding colour stack. Correspondingly, when entering an element, we can obtain all unknowns that were already accessed by previous elements from the respective colour stack.

Once an unknown has been updated by all adjacent elements, we must not put it back onto a colour stack. Otherwise, it would block all elements below it, which still need to be updated. We therefore introduce two additional data structures: an input stream that holds the original unknowns (before their first update access) and

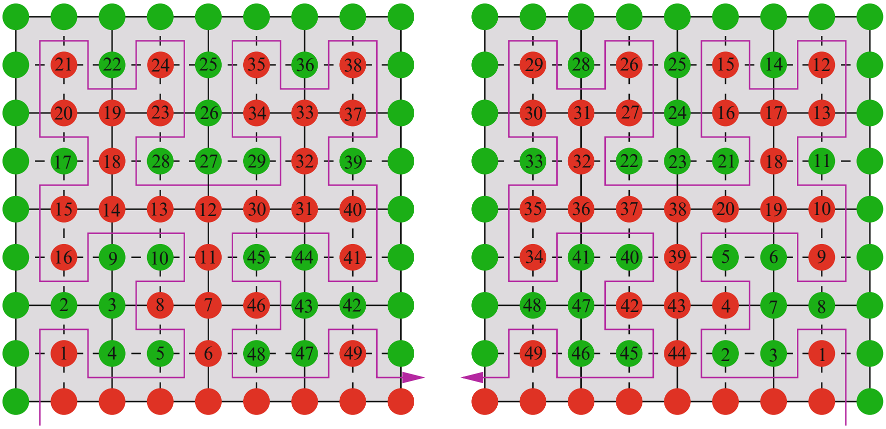


Fig. 14.4 Sequential order of the unknowns in the input and output stream. *Left image*: first-access order or a forward traversal; *right image*: last-access order or a backward traversal

an output stream that holds the finished unknowns, respectively. The order of the unknowns within the input stream has to correspond to the order in which they will be first accessed during the traversal (*first-access order*). Figure 14.4 (left image) illustrates this order for a uniformly refined grid. Note that we will often have a first access to two unknowns at once when entering a cell (3 and 4, or 7 and 8, for example). In that case, the unknown that would be visited first in a refined grid gets the lower number.

In the same way, the output stream will store the unknowns in the order as their processing is finished by the traversal (*last-access order*). This sequential order is illustrated in the right image of Fig. 14.4; however, in that image we do a *backward* traversal! Again, there are frequent situations where two unknowns are finished in the same cell (2 and 3, or 46 and 47, for example). And in concordance with the first-access order, the unknown that would be finished first in a refined grid gets the lower number.

Now compare the two orders resulting from the forward and backward traversal in Fig. 14.4: we can see that the first access during the forward traversal leads to exactly the opposite order that results from the last access during the backward traversal – we will refer to this property as the *inversion property*. Hence, after a forward traversal, all unknowns on the output stack are in a correct order to start a backward traversal – we only need to treat the output stream as a stack, and use it as an input stream in inverted order. The same result is obtained vice versa, so we should treat the input and output stream as input and output *stacks*, as well. After each traversal the input and output stack change their role, and we change the orientation of the traversal.

Figure 14.5, finally, illustrates the overall setting for the traversal algorithm, in particular the interaction of colour stacks, element-oriented processing, and input/output streams. Such stack-based traversal algorithms are possible for several space-filling curves. Figure 14.6 demonstrates the necessary stack property for the standard 2D Peano curve and for the Sierpinski curve.

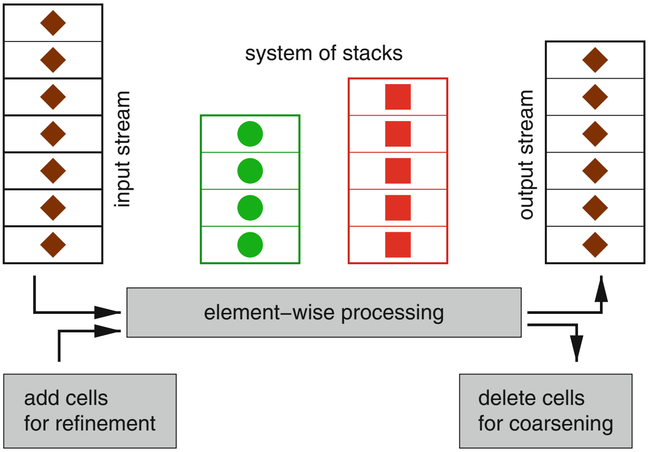

system of stacks
input stream
output stream
element–wise processing
add cells
for refinement
delete cells
for coarsening

Fig. 14.5 System of stacks and streams for the element-oriented traversals

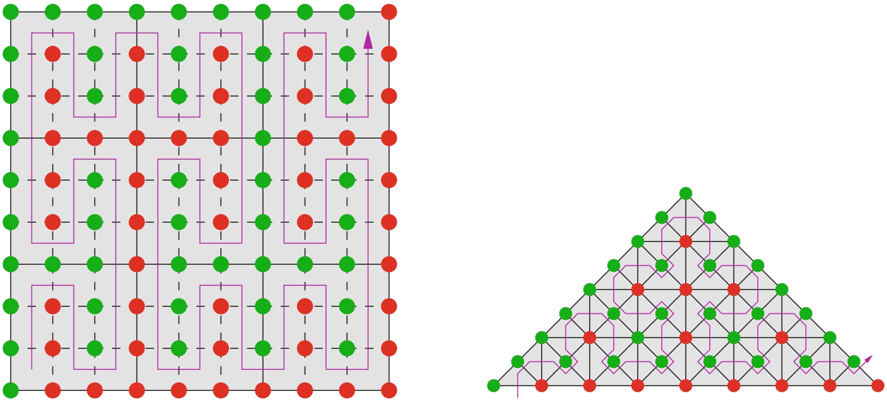


Fig. 14.6 Stack property of the standard 2D Peano curve and of the Sierpinski curve

14.2 Implementation of the Element-Oriented Traversals

In the previous section, we have explained the idea of the element-oriented traversals in enough detail to be able to perform it on a piece of paper – if you want to try it at this point, refer to Exercise 14.1. However, to formulate a proper algorithm that is able to perform the traversals, including the correct stack operations, on a computer, we need to solve the following problems:

- Which unknowns of an element will be taken from the input stream, and which unknowns are already located on a colour stack?

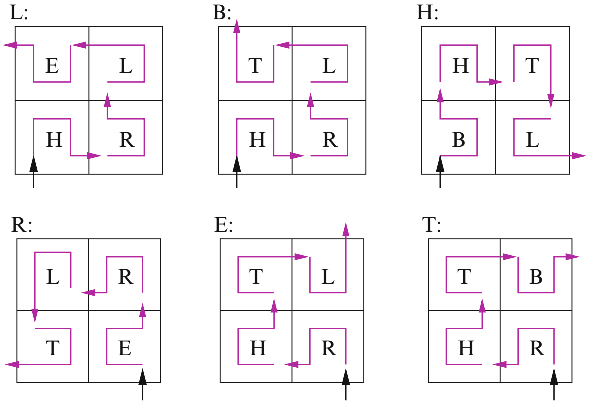


Fig. 14.7 Construction of the turtle grammar for the Hilbert curve

- From which colour stack do we need to retrieve a previously accessed unknown? Similarly, on which colour stack do we need to store an unknown that will be processed further.
- How do we identify unknowns that are finished, i.e. need to be put onto an output stack.

14.2.1 Grammars for Stack Colouring

To decide from which colour stack an unknown needs to be stored or retrieved, we need to classify the unknowns locally into unknowns on the left-hand or right-hand side of the curve. In Sect. 3.4, we discussed the turtle-graphics grammars, which allow us to determine the local course of a space-filling curve within an element, especially the entry and exit edges. Figure 14.7 illustrates the construction of this grammar again, for the Hilbert curve. Once the entry and exit edges of the curve are known, it is easy to decide whether an unknown is located on the left-hand or right-hand side of the curve. Hence, the turtle grammars will be used for our stack-based traversal algorithms.

14.2.2 Input/Output Stacks Versus Colour Stacks

Our next step is to discriminate whether a given unknown should be retrieved from the input stream or from one of the colour stacks. Also, we need to decide whether an updated unknown should be put onto a colour stack or be written to the output stream. To decide this, we require information whether an unknown has been accessed before, which is related to the question which of the adjacent elements has already been visited during a traversal.

One obvious solution is to count the number of accesses to each unknown, which of course has the disadvantage that a separate counter variable is necessary for each

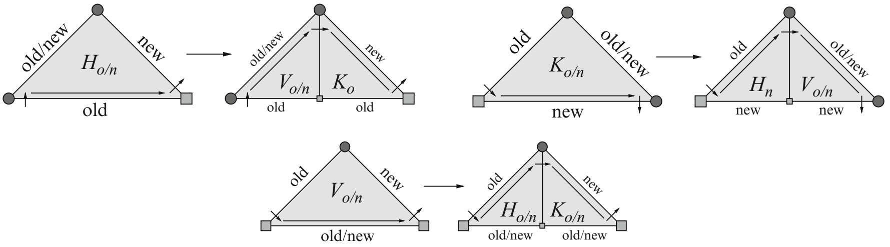


Fig. 14.8 Turtle grammar patterns to determine the node colouring for stack based Sierpinski traversals

unknown.[1] In the following, we will discuss a more elegant approach that is based on an old/new classification of edges. We will present this for a traversal following the Sierpinski-curve on a compatible triangular grid, as the respective construction is easier than for the Hilbert curve.

Stack Colouring for the Sierpinski Curve

We again need a turtle grammar for the Sierpinski curve, which was the topic of Exercise 6.3. In the usual way, we need to examine the entry and exit edges of the Sierpinski curve in a triangular element. As the Sierpinski curve will enter and leave an element always at a node that is adjacent to the hypotenuse, we obtain three different patterns, which are illustrated in Fig. 14.8:

- Pattern H: enter via the hypotenuse, exit across the opposite leg;
- Pattern K: enter via a leg, exit across the hypotenuse;
- Pattern V: enter and exit via the two legs.

As usual, the pattern for the child elements are determined by the patterns of their parents, and follow a fixed recursive scheme.

Old/New Classification

As also illustrated in Fig. 14.8, the patterns H, K, and V provide additional information on whether adjacent elements have already been visited:

- An element adjacent to an entry edge has already been visited – more exactly, it was processed right before the current element;
- An element adjacent to an exit edge was not visited before: it will be processed right after the current element.

---

[1] We assume here that we have exactly one unknown on each corner of the grid. If multiple unknowns would be placed on corners and edges, we only need the respective information for each corner and edge of an element.

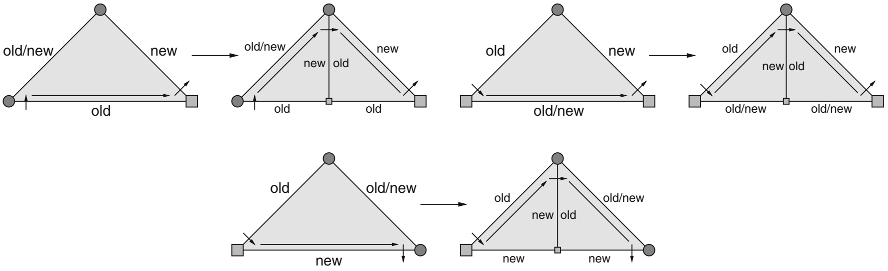


Fig. 14.9 Old/new classification of edges for the Sierpinski traversal

Hence, we only need to determine the old/new information for the third edge (also called the colour edge) in each triangle. As already indicated in Fig. 14.8, we will code this old/new information into the patterns, and use six patterns for the final grammar: H_o, H_n, K_o, K_n, V_o, and V_n. As Fig. 14.9 shows, the old/new information can again be determined from the classification of the parents.

Colour Stacks Versus Input/Output Stacks

With the old/new classification, we can decide whether unknowns on vertices need to be retrieved from the input stream or from one of the colour stacks:

- If an unknown is adjacent to an old edge, it has been processed and updated before, and should therefore be retrieved from the respective colour stack.
- If an unknown is only adjacent to new edges, it was not processed before. It therefore has to be retrieved from the input stream.

Technically, this classification is incomplete: it would be possible that an unknown is adjacent to two new edges, but was processed within an element that is not adjacent to either of these edges and only touches the current element at this node. It can be shown by a study of the possible location and orientation of the parent element that due to the construction of the Sierpinski curve such a scenario does not occur.

After processing an element, we need to decide which unknowns have to be stored on colour stacks – for further use by later elements – and which unknowns can be put onto the output stream, because their processing is finished. Again, a study of the Sierpinski curve's construction leaves only two possible scenarios:

- If an unknown is adjacent to at least one new edge, it will be processed again in the respective adjacent element, and therefore needs to be stored on a colour stack.
- If an unknown is exclusively adjacent to old edges, it will be put onto the output stream.

To decide whether an unknown has been updated by all adjacent elements, it is not immediately obvious that is sufficient that the respective node is connected to two old edges. A pessimistic implementation would introduce a counter for the updates. However, in the case of adaptively refined grids or for complicated geometries, even such a counter would need to be initialised, as the number of adjacent elements can vary between 4 and 8 for a vertex, and is thus not trivial to figure out. It turns out, however, that the two-old-edges criterion is indeed sufficient – which requires a closer study of the scenarios that can occur (see Exercises 14.3 and 14.4, which deal with this problem).

14.2.3 Algorithm for Sierpinski Traversal

For uniformly refined grids, as they result from a fully balanced refinement tree, all ingredients for a fully-functional algorithm are now set. Following the classification shown in Fig. 14.9 – with the element types $H_{o/n}$, $K_{o/n}$, and $V_{o/n}$ – we introduce six separate, nested recursive procedures that either issue calls to the child elements, or, on leaf level, perform the required stack operations and element updates depending on the current cell type. Algorithm 14.1 specifies the respective operations for the case H_n. Algorithm 14.1 is already formulated to work on a bitstream-encoded refinement-tree grid. But does the grid traversal also work on adaptive triangular grids?

Algorithm 14.1: Sierpinski traversal on an element of type H_n

```
Procedure Hn()
    Data: bitstream: bitstream representation of spacetree (streamptr: current position);
          green,red: stacks to store intermediate values
    Variable: entry/exit/angleUnknown: unknowns on the three cell vertices
begin
    // move to next element in bitstream
    streamptr := streamptr + 1;
    if bitstream[streamptr] then
        // obtain local unknowns
        entryUnknown := green.pop();
        angleUnknown := input.pop();
        exitUnknown := red.pop();
        // execute task on current position
        execute(...);
        // execute task on current position
        green.push(entryUnknown);
        green.push(angleUnknown);
        red.push(exitUnknown);
    else
        // recursive call to children
        Vn();
        Ko();
    end
end
```

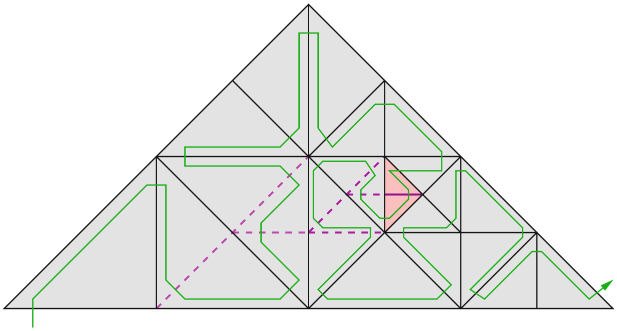


Fig. 14.10 Refinement cascade on an adaptive triangular grid. The *highlighted edge* is marked for refinement; the *dashed edges* mark the forced refinement to retain conformity of the grid

If we compare the stack accesses of an adaptive grid with a uniformly refined grid that has the maximum depth of the adaptive grid, it is easy to check that the stack operations – pop and push from/to colour stacks and input/output stacks – for the unknowns on the adaptive grid cells will stay the same in the full grid. Nor will the order change, in which these accesses occur. Hence, we can interpret an adaptive grid as a regular grid, where a lot of unknowns, and also the corresponding stack accesses are left away. Our stack access scheme will therefore stay correct, if these missing unknowns can be identified by all cells of the grid. This is ensured, if unknowns are only placed on vertices of grid cells, which is always guaranteed in the case of so-called *conforming* grids.

14.2.4 Adaptivity: An Algorithm for Conforming Refinement

An adaptive grid is called *conforming*, if any two adjacent grid cells share either a vertex or an entire edge (for 3D grids: an entire face). Grids with so-called *hanging nodes*, which are placed on an edge of a neighbouring grid cell, are forbidden. Hence, during adaptive refinement of a refinement-tree grid, we have to ensure that such situations are avoided. A natural approach is to mark cells for refinement, first, exchange the refinement information between adjacent cells, and mark additional cells for refinement, if necessary. In that way, we might obtain a refinement cascade, as illustrated in Fig. 14.10.

Again, the algorithm to exchange and update refinement information between grid cells can follow the stack-based scheme, and works similar to Algorithm 14.1. As the exchanged unknowns now reflect the refinement status of edges, unknowns are now located on edges instead of on vertices, which even simplifies the stack access, as all unknowns are accessed exactly twice. Algorithm 14.2 outlines the scheme for the case H_o.

If we examine Fig. 14.10 more closely, we can see that the refinement cascades can propagate in the direction given by the Sierpinski order, but also against it. Even worse, the propagation can change between these orientations multiple times.

Algorithm 14.2: Sierpinski traversal to propagate refinement information (for element type H_o)

```
Procedure Ho()
    Data: bitstream: bitstream representation of spacetree (streamptr: current position);
          green,red: stacks to store refinement status;
          input,output: streams for refinement status of edges
    Variable: left/rightLeg, hypotenuse: refinement of the three cell edges
begin
    // move to next element in bitstream
    streamptr := streamptr + 1;
    if bitstream[streamptr] then
        // obtain local unknowns
        leftLeg := green.pop();
        rightLeg := input.pop();
        hypotenuse := red.pop();
        // update refinement status of current cell
        execute(...);
        // execute task on current position
        output.push(leftLeg);
        green.push(rightLeg);
        output.push(hypotenuse);
    else
        // recursive call to children
        Vo();
        Ko();
    end
end
```

As Algorithm 14.2 can only propagate the information in one of the directions, we require multiple traversals – in practice, we repeat the traversals until a static refinement status is reached. Hence, Algorithm 14.2 is not efficient for refinement at singular points. Instead, it is intended for scenarios where a large part of the grid cells are refined or coarsened at the same time. In practice, we should therefore collect all refinement and coarsening decisions for all grid cells, for example as part of an error estimation traversal, and then process all refinement and coarsening requests together.

14.2.5 A Memory-Efficient Simulation Approach for Dynamically Adaptive Grids

The stack&stream-based approach presented in this chapter combines a series of individual steps that are based on each other and can thus achieve their full advantage only if all steps are combined:

- The bitstream representation for the refinement tree grids leads to minimal memory requirements to store the adaptive grids.

- The stack&stream-based traversal algorithms ensure that information can be efficiently exchanged between adjacent cells, even though no neighbour information is permanently stored.
- The traversals require an element-oriented discretisation and element-oriented processing of operators (incl. residual computation, etc.). In that way, an explicit set-up of global system matrices is avoided, which again leads to substantial gains in memory requirement. A prerequisite is, however, that the element stiffness matrices can be efficiently computed for each cell, or that only a couple of different template matrices need to be stored (which is possible, if a geometrically uniform refinement of grid cells is used).
- The restriction to stacks and streams as basic data structures is highly cache friendly. Together with the locality properties introduced by the space-filling curves, we obtain excellent cache performance for the respective traversal algorithms.
- The traversals are fully compatible with the parallelisation approaches discussed in Chap. 10.

The approach is therefore most efficient in simulation scenarios, where dynamically adaptive grids are used, i.e., grids that need to be refined and coarsened throughout the entire simulation. Such refinement and coarsening can, for example, be triggered by frequent changes of the computational domain (consider fluid flow simulation in moving geometries) or by numerically motivated refinement at moving hot-spots (consider refinement along a propagating shock wave, for example). In such cases, the frequent refinement and coarsening of grids is already a challenge for the data structure, as locality properties cannot be retained by classical approaches. In addition, an explicit assembly of global matrices is not feasible, if such an assembly has to be repeated in almost every time step. For application scenarios, as well as for further details on the design, implementation, and integration of such stack&stream-based approaches, see the references section at the end of this chapter.

14.3 Where It Works: And Where It Doesn't

The stack&stream-based traversal and data processing on adaptive grids requires a couple of properties that need to be fulfilled by the space-filling element order:

- The *stack property* simply states that data can be put to stacks and retrieved from stack by adjacent elements during traversals. In 2D, this property is satisfied by all edge-connected space-filling curves – in particular, for Hilbert, Peano, and Sierpinski curves. It does not apply for Morton order, however (see Sect. 14.3.2).
- In 3D and higher dimensions, the stack property requires that at a common (hyper-)face of two subdomains (resulting from the curves construction process) the element orders imposed by the space-filling curve on the two faces are exactly inverse to each other. We will refer to this as the *palindrome property*. To have

curves matching at common faces in this way seems to require that the curve is face-connected, however, this is not sufficient!

- The *inversion property* ensures that data written to the output stream is required in opposite direction by the input stream during the following, backward traversal. Hence, the last-access order of the forward traversal has to be inverse to the first-access order of the backward traversal (and vice versa). The inversion property also applies to Hilbert, Peano, and Sierpinski curves in 2D.

For the 3D and higher-dimensional case, the situation is, in general, much more complicated than in 2D. 3D Hilbert curves do no longer satisfy the stack property (see Sect. 14.3.1) – in particular they do not satisfy the palindrome property. For Sierpinski curves, already the construction of a face-connected curve is difficult. However, for the 3D Sierpinski curve introduced in Sect. 8, a stack&stream-traversal is possible [121] (see also Exercise 14.6). Peano curves (the canonical curves of switch-back type, only) have been proven to be ideal for stack&stream-traversals. They satisfy the palindrome property, which is also a consequence of the projection property of the Peano curve, i.e., the fact that projections along the coordinate directions lead to lower-dimensional Peano curves. Stack-based Peano traversal can thus be constructed for n-dimensional spacetree grids (see the references of this chapter). In the following, we discuss the Morton-order and 3D-Hilbert case, in order to illustrate the different roadblocks to obtaining stack&stream traversals.

14.3.1 Three-Dimensional Hilbert Traversals

Figure 14.11 illustrates the examination of the stack property for the first 3D Hilbert iteration. In each of the three images, the unit cube is split along a cutting plane that is parallel to the coordinate planes. The palindrome property is only satisfied by the cutting plane in the top-left image. For the front-back cutting plane, as in the lower left image, the Hilbert order is parallel, but not in reverse order. For the horizontal cutting plane in the right image, the traversal orders prescribed by the Hilbert curve in the adjacent cells are not even parallel to each other. As the basic patterns already fail, none of the multiple Hilbert curve variants is able to satisfy the stack property.

14.3.2 A Look at Morton Order

For Morton order (or the Lebesgue curve), as introduced in Sect. 7.2, the stack condition between adjacent planes quite obviously does not apply. However, as we can see from Fig. 14.12, the bit representations of adjacent cells match in the sense that adjacent cells have identical bits for the coordinate directions of the cutting plane. This projection property is a direct result of the index construction, where the bit representations for the individual coordinate directions are simply interleaved

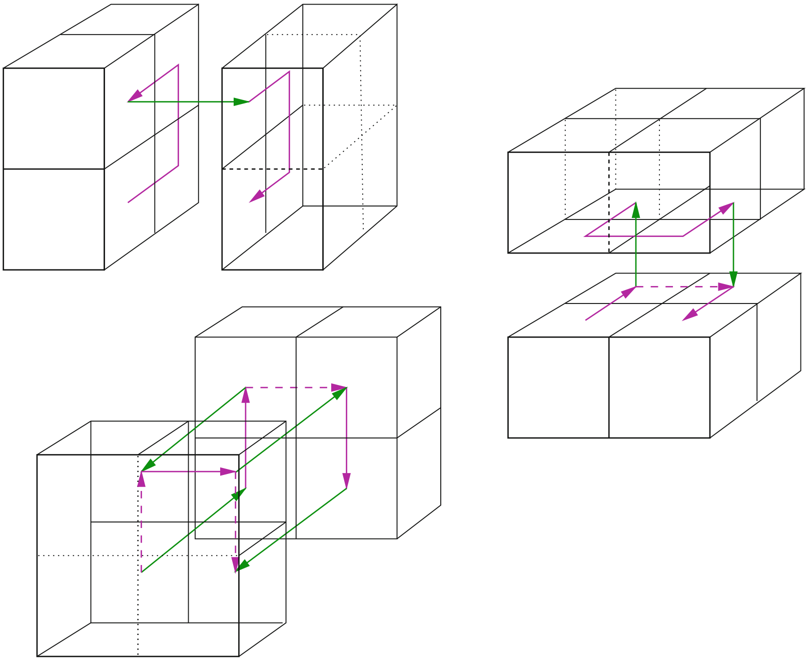
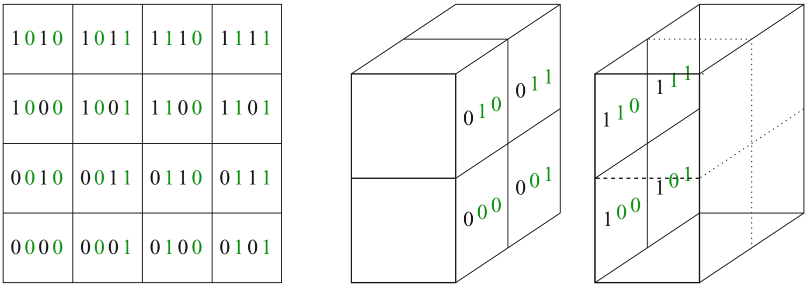


Fig. 14.11 Examination of the palindrome property of the 3D Hilbert curve: the stack property is only satisfied at the cutting plane shown in the *top left* image. *Dashed arrows* indicate the relative traversal order within a 2 × 2-slice



1010	1011	1110	1111
1000	1001	1100	1101
0010	0011	0110	0111
0000	0001	0100	0101

Fig. 14.12 Bit representation of Morton indices along in 2D and 3D

and do not depend on each other. Hence, this property stays valid for n-dimensional Morton order, as well.

The matching indices suggest that we might again be able to exchange data on common hyperfaces via simple data structures, which would no longer be stacks, but streams or queues, instead, to match the index properties. However, such a data structure does not work in the expected way, as illustrated in Fig. 14.13. Imagine a queue data structure for edge-based data in 2D, as in the left image of Fig. 14.13.

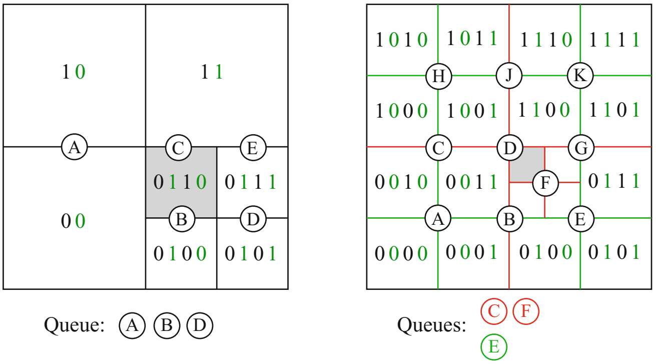


Fig. 14.13 Violation of potential queue properties for Morton order

There, we plan to append edges at the tail of the queue, such that they can be used by the adjacent cell. Data is retrieved from the head of the queue. During traversal of the first three elements, we will first obtain the edge data A, B, and D from an input stream, and append them to the queue after leaving the elements. In the highlighted fourth cell, we would read C from the input stream, but cannot obtain B from the queue, as it is blocked by A, which is still at the head of the queue. Note that our idea of using two queues (or stacks) for unknowns to the left and right of the curve breaks down for Morton order: the unknowns A, B, and D are left of the curve when stored on the queue, but are right of the curve when they are obtained. Hence, according to a left-right colouring of queues, they should be placed on the same queue – however, our example shows that the resulting queue scheme already fails for the simplest adaptive example.

In the right image of Fig. 14.13, we try an alternate, level-oriented colouring scheme and use vertex-located data. According to this scheme, unknown C would be stored in the red queue after traversal of the first four elements. In elements 0100 and 0101, B and E would be read from input and stored to the green queue. As C, placed in the red queue, does not obstruct access to B and E, we can easily retrieve them in the next two fine-level elements. During these two cells, we would also append F to the red queue. Thus, we are faced with a queue conflict in the highlighted cell, as F should be retrieved from the red queue, which still has C at its head.

Inversion Property for Input/Output-Streams in Morton Order

Note that – despite the in-order projection property of Morton order – the input and output stream do not obey a queue order: the processing of F is obviously finished before C and D have been updated by all adjacent cells. Hence, while C and D are read before F by the input stream (i.e., come before F in first-access order), they are written to an output stream after F (i.e., come later in last-access order). In that respect, queues seem to be inherently incompatible to space-filling curves as data structures, which is apparently related to the recursive construction of space-filling curves: recursion is an approach that is naturally connected to stack data structures.

If we examine the right image of Fig. 14.13 more closely, we can find that an input-output-stream of the node data A–K would actually follow a stack principle, if we alternate between forward and backward traversal – such that the inversion property is satisfied. To verify this, we set up a table that lists the first-access and last-access order for the node data during forward and backward traversal:

	Forward	Backward
First access	A B C D E F G H J K	K J G D H C E F B A
Last access	A B F E C H D G J K	K J H G F E D C B A

Note that the diagonal sequences are exactly inverse to each other. Hence, after a forward traversal that has written all data to a stream in last-access order – i.e., simply writes all data to a stream once it has been processed by all elements – a backward traversal can use this stream in reverse order and will find all elements in correct first-access order.

Hence, to find an effective stream-based traversal scheme for spacetrees stored in Morton order, we would require an efficient solution for the intermediate storage of node- and edge-based data. For example, we could use a lot more data streams – one for each level of recursion – or invest $(d-1)$-dimensional grids to exchange data by random access. It is open, however, whether such solutions would be computationally efficient. An alternative could be to store the interfaces between d-dimensional spacetrees explicitly and exchange data via these permanent data structures. Such *boundary-extended spacetrees* have been suggested by Frank [89].

References and Further Readings

Cache oblivious approaches based on refinement trees, space-filling-curve orders, and stack-oriented access have been developed for spacetree grids and Peano-traversals by Zenger, Guenther [114], Mehl et al. [183], and Weinzierl and Mehl [266]; for triangular grids and Sierpinski orders, the approaches were introduced by Bader and Zenger [23, 25]. The approaches have been applied to scenarios where dynamically adaptive refinement and coarsening of the discretisation grid is required either by a changing computational domain or by numerical requirements, such as in computational fluid dynamics [52, 181] (in particular, fluid-structure interaction [48, 180]) or in the context of Tsunami simulation [20, 38]. Parallelisation of the approaches has been studied by Weinzierl et al. [53, 182, 265], in particular. The extension of the Peano approach to higher dimensions is presented in [120], where the authors also discuss the palindrome property and the Peano curve's projection property as requirements to construct a stack-based scheme.

In the references section of Chap. 13, we have already discussed works that exploit the locality properties of space-filling curves not only for parallelisation, but also to improve cache efficiency. In the context of molecular simulation, such approaches have been studied by Hu et al. [134] and by Mellor-Crummey et al. [184] – the latter examined both molecular dynamics applications and an unstructured-mesh code. For related work on general graph-structured computations, see also [7, 205]. A comparison of different re-ordering methods can be found in [119]. Behrens et al. [36] used Sierpinski curves on bisection-generated adaptive grids and observed improvements regarding the cache performance, but also regarding the behaviour of the ILU-based preconditioner, which was able to profit from a space-filling-curve numbering of the unknowns.

In computer graphics, the stack&stream approach is similar to the usage of triangle and tetrahedral strips – see the references section of Chap. 9. In that context, the group of Gotsman [27, 44] introduced and analysed cache-oblivious algorithms based on triangle strips, providing lower bounds for the number of cache misses that are obtained during traversals and for different cache sizes. A further work that focuses on caches was presented by Hoppe [133]. Gerstner [97] presented an approach that is most closely related to our stack&stream approach – he studied algorithms based on newest-vertex-bisection and respective binary refinement trees, using the Sierpinski curve to generate triangle strips. He introduced stack structures, quite similar to the approach discussed here, to exchange data between neighbours, but required a logarithmic number of stacks.

What's next?

- Actually, this was the last major chapter of the book. The next and final chapter will list some further areas of applications for space-filling curves.
- Did you skip any chapters so far? Well, now would be the right time to go back…

Exercises

14.1. Take a piece of paper and try to perform the traversal algorithms step by step on simple quadtree grids. In particular, pay attention to the stack operations. Try to do the same exercise using the Sierpinski curve (on a compatible, uniformly refined triangular grid) or the Peano curve (on a $3^k \times 3^k$ Cartesian grid).

14.2. Sketch an algorithm that exchanges the indices of neighbouring quadtree cells via a stack-based traversal.

14.3. Consider a triangle that has two *old* edges adjacent to the right angle, and examine the possible scenarios for the old/new classification of the adjacent triangles and of the respective parent cells. Is it guaranteed that the node at the right angle has already been processed by all adjacent elements (esp. the triangle that only touches the vertex, but not the two edges at the right angle)?

14.4. Now consider the case that two *old* edges are adjacent to the entry node of a triangle (note that such a scenario can only occur for triangles of type V_o or H_o). Use Fig. 14.7 to determine the possible parent and grandparent triangles – is it possible that the entry node touches a "new" edge?

14.5. For adaptive quadtree grids, it can be advantageous to require a certain size balance between adjacent grid cells. For example, we can request that grid cells that share a common boundary may have a size difference of at most 2:1 (2:1 balancing). Modify the algorithm for conforming refinement, as discussed in Sect. 14.2.4, to obtain such a 2:1 balanced quadtree.

14.6. Examine the 3D Sierpinski curve introduced in Sect. 8 and check whether it satisfies the palindrome property.

Chapter 15
Further Applications of Space-Filling Curves: References and Readings

In 1981, Goldschlager – in his short communication on a simpler recursive algorithm for Hilbert and Sierpinski curves [101] – wrote that

> The main application of these curves to Computer Sciences has been recreational, as they produce pretty pictures on a plotter.[1]

And even though being contradicted already in a 1983 article by Witten and Wyhill [274] in the same journal, Goldschlager was certainly right in the sense that for a long time space-filling curves where "topological monsters" that were of interest to a few mathematicians, but of little practical use.

In the meantime, however, applications have come up in various problem settings. Already in 1988, Bartholdi and Platzman [32] formulated a so-called *Generic Space-filling Heuristic* (GSFH), which summarises almost any application of space-filling curves in two short sentences:

Definition 15.1 (Generic Space-filling Heuristic [32]).

1. Transform the problem in the unit square, via a space-filling curve, to a problem on the unit interval.
2. Solve the (easier) problem on the unit interval.

In this monograph, we only discussed applications of space-filling curves in scientific computing, and even within this discipline, we concentrated on aspects regarding the connection of space-filling curves to quadtrees and octrees (in Chap. 9) and their use in parallelisation (Chap. 10) and for cache-efficient algorithms (Chaps. 13 and 14). Hence, in this last chapter, we will provide a short overview on further applications, with a stronger focus on classical computer science.

---

[1] One of the earliest applications of space-filling curves was actually published in a series on "Computer Recreations" and carried the title *Space-filling curves, or how to waste time with a plotter* [198]. It contained one of the earliest algorithms to draw a space-filling curve, and mentioned that "drawing the polygons is an excellent test of the long-term positional stability of your plotter".

M. Bader, *Space-Filling Curves*, Texts in Computational Science and Engineering 9,
DOI 10.1007/978-3-642-31046-1_15, © Springer-Verlag Berlin Heidelberg 2013

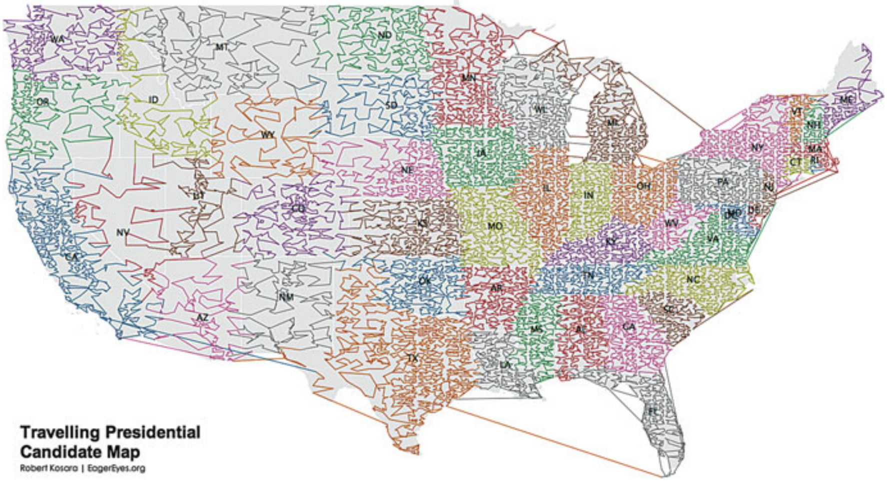


Fig. 15.1 "Travelling Presidential Candidate Map" – a tour through all 37,000 ZIP code locations in the United States, computed using the Hilbert curve (special thanks go to Robert Kosara, who generated this graph and permitted its use in this book; for a detailed explanation of this image, see his website: http://eagereyes.org/Applications/ZIPTPCMap.html)

Space-Filling Curves As Heuristics for Combinatorial Problems

Bartholdi and Platzman [32] suggested their general space-filling curve heuristics for several combinatorial problems in scientific computing, such as the travelling salesman problem, or the K-Median problem. In the travelling salesman problem, the shortest path is wanted to visit a set of given locations exactly once – an excellent illustration of the problem is given in Fig. 15.1. It reproduces Robert Kosara's "Travelling Presidential Candidate Map", which shows a tour through all 37,000 ZIP locations in the United States – the tour is an approximate solution to the traveling salesman problem using the Hilbert curve as heuristics. The traveling salesman problem is known to be NP-complete, but approximations to the solution can be found quickly via computing the space-filling-curve order of the given point set [31,219]. Bertsimas and Grigni [41] provided an example that the solution found via this heuristics can be longer than the exact solution by up to a logarithmic factor. An analysis of the tour length obtained via the space-filling curve heuristics was given by Gao and Steele [94]. In general, the space-filling-curve heuristics excels by the quick computation of the tour, and can serve, for example, to compute an initial solution for more sophisticated algorithms (e.g., [229]). As a testbed to study properties of the traveling salesman problem and heuristic algorithms to solve it, Norman and Moscato used Sierpinski curves to generate problem instances with controlled properties [197].

The K-Median problem is a variant of a partitioning problem: for a given set of points, K “medians” are to be found that minimise the sum of the distances of all points to their respective closest median. In addition to Bartholdi’s algorithm in [32], approaches based on space-filling curves were used in the context of classification, for example for vector quantisation in signal processing [83] or for fuzzy controllers [82], but also for the regular nearest-neighbour problem [66, 161, 173], and for the similar problem of contact searching [77].

Data Bases and Geographical Information Systems

Space-filling orders were also exploited to construct data structures for range queries on images [18, 140]. These use cases are closely related to applications of space-filling curves in geographical information systems [1, 170], geographical data bases, and data bases in general. Range queries on geometrical or geographical data – idealised as multidimensional point sets, for example – led to application of space-filling curves in these fields. Early works were, for example, based on locational codes [2] (similar to Z order) or bit interleaving [204] (Morton order). A review on space-filling-curve and other techniques is given in [51]. In data bases, keys consisting of multiple attributes can be treated as such multi-dimensional point data [86, 139]. Hilbert curves and Morton/Z-order, in particular, have been widely used in the context of data bases to extent classical 1D indexing data structures for higher dimensions [154, 257]. Examples include kd-trees [156], R-trees [16, 122, 123, 146], or UB-trees [35, 226] (which combine B-trees with Z-order). The use of quadtree and octree data structures was already outlined in the references for Chap. 9.

Related to the use in data bases is the application of space-filling curves for storage of multidimensional data, in general. such as for geographical point data [262], for image data bases [203], but also for systems to explore simulation data, e.g. resulting from turbulence simulation [216] or from simulations on unstructured grids [208].

Signal Processing and Computer Graphics

The earliest applications of space-filling curves appeared in signal processing, for example by Abend et al. [3] in 1965, who used space-filling-curve mappings for pattern classifications. Their technique to exploit $n : 1$ and $1 : n$ mappings to encode and decode multidimensional signals to 1D signals and apply 1D methods, was also used by Bially [42] in the context of bandwidth reduction. The work of Lempel and Ziv [160], who used a 2:1 mapping with 1D techniques for data compression (esp. image data), also falls in this category.

In computer graphics, space-filling curves appeared as data structures or scanning orders (“Peano scan”) for images for various applications, such as image compression [15, 160, 278] (space-filling scans followed by 1D techniques), colour reduction [250], video scrambling [175], lossless or lossy image coding [71, 185, 215], or even digital watermarks [192]. Space-filling scans were also suggested for dithering of images, such as by Witten and Neal [273] for the bilevel (black&white) display of continuous tone images, or for image halftoning in general [259, 282]. For the successive generation of Delauney triangulations from given point sets [11], space-filling curves were used to order the points to improve the quality of the Delauney meshes [50,171,283]. Further applications are texture classification [158], topological matching [251], or image browsing [70].

As in the scientific-computing applications, space-filling curves were integrated with quadtrees, octrees, and adaptively refined triangular meshes to obtain data structures for computer graphics and geometry processing. These were already outlined in the references for Chaps. 9 and 14. The combination with structured adaptive refinement of meshes – with a wide range of use cases in computer graphics and scientific computing – is still of high relevance, and will hopefully drive further research on space-filling curves.

What's next?

- That's all for now – you've reached the end of the book. In the appendix, you will find solutions to selected exercises (including some further interesting material).
- Did you do all (or at least most of) the exercises in the previous chapters? Well, if you didn't, then don't proceed to the solutions . . .

Appendix A
Solutions to Selected Exercises

A.1 Two Motivating Examples

(no exercises)

A.2 Space-Filling Curves

2.1 The corner points of the approximating polygons reflect an entry/exit point of the curve in each subsquare. As an entry point, the respective points – by construction – will always be located in the first subsquare and correspond to the leftmost of the respective nested intervals. Hence, the entry point is an image of the left boundary of the respective interval. As examples, we obtain the parameter–point pairs $0 \to (0,0)$, $\frac{1}{4} \to (0, \frac{1}{2})$, etc.

2.2 Some further variants to construct a curve similar to Moore's variant are given in Fig. A.1 (see also [168]).

2.3 The proof of continuity of the Peano curve can be copied almost word by word from the respective proof for the Hilbert curve, as given in Sect. 2.3.5: For two given parameters t_1 and t_2, we choose a refinement level n, such that $|t_1 - t_2| < 9^{-n}$ (owing to the substructuring into nine subsquares in each step); t_1 and t_2 are then mapped to points $p(t_1)$ and $p(t_2)$ that are lying in two adjacent subsquares of side length 3^{-n}. Their distance $\|p(t_1) - p(t_2)\|$ thus has to be smaller than $3^{-n} \cdot \sqrt{5}$ (and the rest of the proof is straightforward).

2.4 I hope you didn't take that exercise too seriously…

M. Bader, *Space-Filling Curves*, Texts in Computational Science and Engineering 9,
DOI 10.1007/978-3-642-31046-1, © Springer-Verlag Berlin Heidelberg 2013

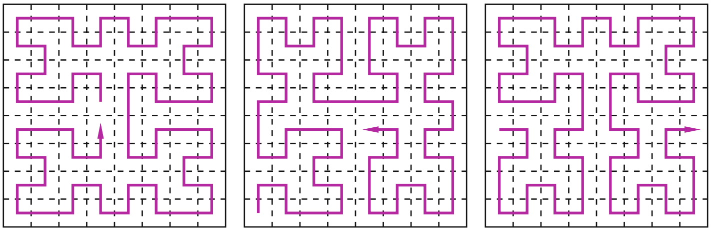


Fig. A.1 Three variants of the Hilbert-Moore curve. The left-most variant uses the same orientation of the four Hilbert parts as Moore's curve, but moves the start and end point to the centre of the unit square

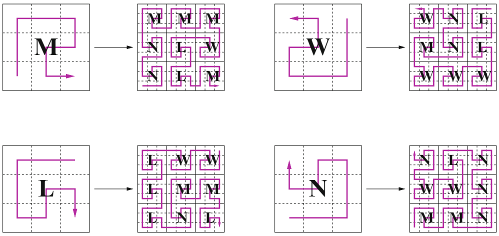


Fig. A.2 Construction and grammar symbols of the Peano-Meander curve

A.3 Grammar-Based Description of Space-Filling Curves

3.1 The derivation of a grammar for the Peano-Meander curve is illustrated in Fig. A.2. We need the following symbols and production rules:

- Non-terminals: $\{M, W, L, N\}$, start symbol M
- Terminals: $\{\uparrow, \downarrow, \leftarrow, \rightarrow\}$
- Production rules:

$$
\begin{aligned}
M &\longleftarrow N \uparrow N \uparrow M \rightarrow M \rightarrow M \downarrow W \leftarrow L \downarrow L \rightarrow M \\
W &\longleftarrow L \downarrow L \downarrow W \leftarrow W \leftarrow W \uparrow M \rightarrow N \uparrow N \leftarrow W \\
L &\longleftarrow W \leftarrow W \leftarrow L \downarrow L \downarrow L \rightarrow N \uparrow M \rightarrow M \downarrow L \\
N &\longleftarrow M \rightarrow M \rightarrow N \uparrow N \uparrow N \leftarrow L \downarrow W \leftarrow W \uparrow N
\end{aligned}
$$

3.2 From Fig. 2.5, it is straightforward to obtain the production rule for the start symbol (here: M):

$$M \longleftarrow \bar{B} \uparrow \bar{B} \rightarrow \bar{A} \downarrow \bar{A},$$

where the non-terminals $\bar{A}$ and $\bar{B}$ correspond to the patterns represented by A and B in the regular Hilbert-curve grammar (as in Fig. 3.1). However, their orientation is exactly inverse. This also reflects in the productions for $\bar{A}$ and $\bar{B}$:

$$\begin{aligned} \bar{A} &\longleftarrow \bar{C} \rightarrow \bar{A} \uparrow \bar{A} \leftarrow \bar{H} \\ \bar{B} &\longleftarrow \bar{H} \leftarrow \bar{B} \downarrow \bar{B} \rightarrow \bar{C} \end{aligned}$$

(and similar for $\bar{H}$ and $\bar{C}$). Thus, the Hilbert-Moore grammar requires five non-terminals, but M is only used as start symbol.

3.3 An algorithm for matrix-vector multiplication using Hilbert traversal is discussed in Sect. 13.2, see in particular Algorithm 13.3.

3.5 If rotation is neglected, all basic patterns of the Hilbert curve come down to the two patterns ↑ ↱ ↑ ↱ ↑ and ↑ ↰ ↑ ↰ ↑. For these, two non-terminals, in the following denoted R and L, are sufficient, with terminal productions

$$R \longleftarrow \uparrow \Rsh \uparrow \Rsh \uparrow \qquad\qquad L \longleftarrow \uparrow \Lsh \uparrow \Lsh \uparrow .$$

Thus, we have to include the turns and moves between the patterns into the non-terminal productions:

$$\begin{aligned} R &\longleftarrow \Rsh L \Rsh \uparrow R \Lsh \uparrow \Lsh R \uparrow \Rsh L \Rsh \\ L &\longleftarrow \Lsh R \Lsh \uparrow L \Rsh \uparrow \Rsh L \uparrow \Lsh R \Lsh \end{aligned}$$

Compare Fig. 3.6 on page 39 for illustration. From the structure of the productions, it is clear that a turtle that obeys to this grammar might do multiple turns before performing the next forward step.

A.4 Arithmetic Representation of Space-Filling Curves

4.1 With $8 = 20_4$ and $5 = 11_4$, the quaternary representation of the fractions $\frac{1}{8}$, $\frac{1}{3}$, and $\frac{2}{5}$ can be computed in a regular long division, as learned in school for the decimal system:

$$\begin{array}{lll} 1 : 20_4 = 0_4.02 & 1 : 3_4 = 0_4.11\ldots & 2 : 11_4 = 0_4.\overline{12}\ldots \\ \underline{0} & \underline{0} & \underline{0} \\ 10 & 10 & 20 \\ \underline{\ \ 0} & \underline{\ \ 3} & \underline{11} \\ 100 & \ \ 10 & \ \ 30 \\ \underline{100} & \ \ \ \underline{3} & \ \ \underline{22} \\ \ \ - & \ \ \ 1\ldots & \ \ \ 2\ldots \end{array}$$

Thus, we have $\frac{1}{8} = 0_4.02$, $\frac{1}{3} = 0_4.11\ldots$, and $\frac{2}{5} = 0_4.\overline{12}\ldots$

4.2 $\frac{1}{3}$ has the quaternary representation $0_4.111\ldots$ (see Exercise 4.1). The recursion equation (4.1) for the Hilbert mapping thus reads:

$$h\left(\frac{1}{3}\right) = h(0_4.111\ldots) = H_1 \circ h(0_4.11\ldots) = H_1 \circ h\left(\frac{1}{3}\right), \tag{A.1}$$

which means that $h\left(\frac{1}{3}\right)$ is a fixpoint of operator H_1. The fixpoint equation

$$H_1\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} 0 \\ \frac{1}{2} \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix},$$

is solved by $x = 0$ and $y = 1$, such that $h\left(\frac{1}{3}\right) = (0, 1)$.

For $\frac{2}{5}$ (or $0_4.121212\ldots$), Eq. (A.1) turns into

$$h\left(\frac{2}{5}\right) = h(0_4.121212\ldots) = H_1 \circ H_2 \circ h(0_4.1212\ldots) = (H_1 \circ H_2) \circ h\left(\frac{2}{5}\right).$$

Hence, $h\left(\frac{2}{5}\right)$ can be computed as the fixpoint of the operator $H_1 \circ H_2$ (see Exercise 4.6).

4.4 Analogous to Eq. (4.3) on page 50, we obtain the following arithmetisation for the Hilbert-Moore mapping $m(t)$,

$$m(0_4.q_1q_2q_3\ldots) = \lim_{n\to\infty} M_{q_1} \circ H_{q_2} \circ H_{q_3} \circ \cdots \circ H_{q_n}\begin{pmatrix} 0 \\ 0 \end{pmatrix},$$

where we need to derive a new set of operators M_i to reflect the fact that, in the first recursion step, we assemble four regular Hilbert curves, but using different orientation. The operators M_i are:

$$M_0 := \begin{pmatrix} 0 & -\frac{1}{2} \\ \frac{1}{2} & 0 \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} \frac{1}{2} \\ 0 \end{pmatrix} \qquad M_1 := \begin{pmatrix} 0 & -\frac{1}{2} \\ \frac{1}{2} & 0 \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} \frac{1}{2} \\ \frac{1}{2} \end{pmatrix}$$

$$M_2 := \begin{pmatrix} 0 & \frac{1}{2} \\ -\frac{1}{2} & 0 \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} \frac{1}{2} \\ 1 \end{pmatrix} \qquad M_3 := \begin{pmatrix} 0 & \frac{1}{2} \\ -\frac{1}{2} & 0 \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} \frac{1}{2} \\ \frac{1}{2} \end{pmatrix}$$

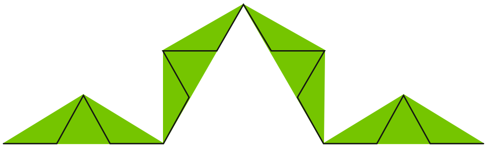
4.6 As an example, we compute the operator $H_{12} = H_1 \circ H_2$:

$$\begin{aligned} H_1 \circ H_2\begin{pmatrix} x \\ y \end{pmatrix} &= \begin{pmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{pmatrix}\left[\begin{pmatrix} \frac{1}{2} & 0 \\ 0 & \frac{1}{2} \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} \frac{1}{2} \\ \frac{1}{2} \end{pmatrix}\right] + \begin{pmatrix} 0 \\ \frac{1}{2} \end{pmatrix} \\ &= \frac{1}{4}\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} \frac{1}{2} \\ \frac{3}{4} \end{pmatrix}. \end{aligned}$$

As there are only four orientations of the basic Hilbert pattern, the matrix part of the operators (corresponding to the rotation/reflection) will only have four different values (up to scaling).

A.5 Approximating Polygons

5.1 The Koch curve can be enclosed by triangular areas in the following way:



On refinement level n, each triangle has a basis of length 3^{-n}, a height of $3^{-n}\frac{\sqrt{3}}{6}$, and thus an area of $3^{-2n}\frac{\sqrt{3}}{12}$. Each green triangle is split into four smaller ones from each level to the next. Thus, there are $4^{-n} = 2^{-2n}$ triangles on the n-th level, with a total area of $2^{-2n} \cdot 3^{-2n}\frac{\sqrt{3}}{12} = \left(\frac{2}{3}\right)^{-2n}\frac{\sqrt{3}}{12}$, which converges to 0 for $n \to \infty$. As the Koch curve is enclosed by the green triangles on all levels, its area has to be even smaller, such that it can only be 0.

5.3 Figure A.3 shows subsequent iterations for two Koch curves, where the "middle third" is replaced by a very narrow isosceles triangle. Cesaro showed, in 1905 [64], that if the acute angle approaches 0, the curve becomes space-filling – compare the approximating polygon of the Sierpinski curve (compare Fig. 6.3).

5.4 The "turtle" grammar is quite simple, as it only requires a single non-terminal:

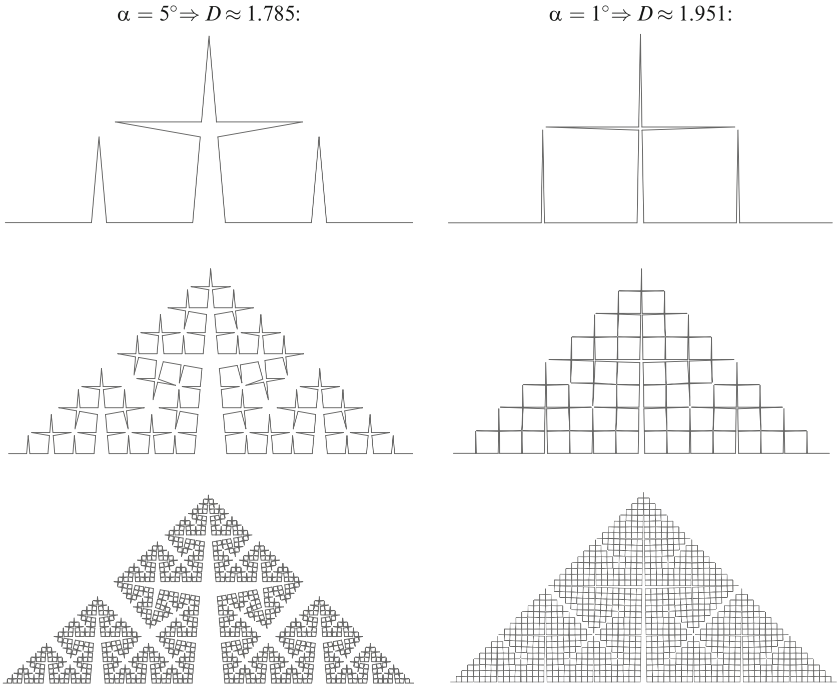
$$\begin{aligned} K \longleftarrow\ & K\, l\, K\, r r\, K\, l\, K \\ \mid\ & \uparrow \end{aligned}$$

(l and r are terminal symbols that represent a left or right rotation by 60°).

In contrast, deriving a plotter grammar for the Koch curve is very tedious, as the "baseline" of the curve can occur in all 60°-steps.

5.5 The construction of the grammars should be no problem, however, it is worth to state that a "turtle" grammar with only one non-terminal will not work, as the generator has to be applied in two different orientations (first "turn left" or first "turn right"). Hence, the generator-based approach to construct fractal curves only works, if we allow such variations of the generator.

5.6 For the canonical Peano curve, the respective generator is applied in two symmetric orientations. Figure A.4 shows the second iteration and polygon of a



$\alpha = 5° \Rightarrow D \approx 1.785$:

$\alpha = 1° \Rightarrow D \approx 1.951$:

Fig. A.3 A Koch curve approximating the Sierpinski curve

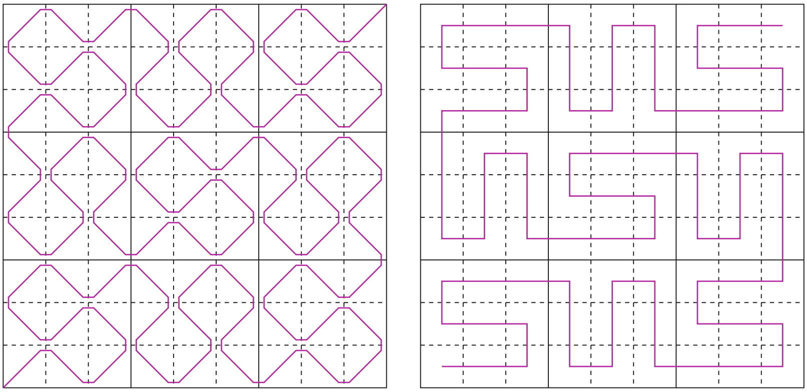
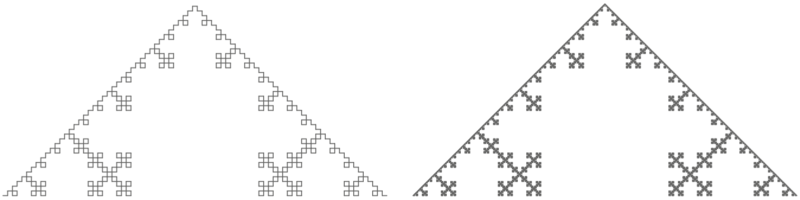


Fig. A.4 Iteration and approximating polygon of the Peano curve required by Exercise 5.6

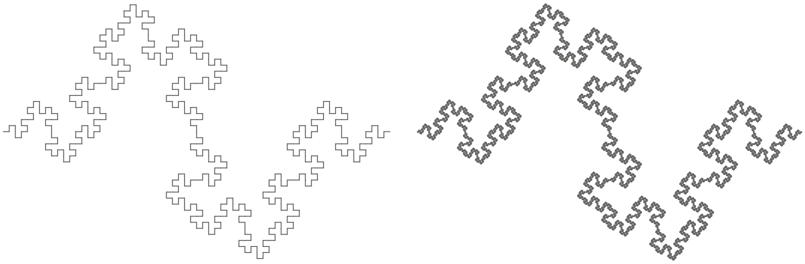
curve where the generator is uniformly oriented (the first turn in a subsquare is always to the right).

5.7 Iterations of the resulting curves are plotted in Fig. A.5. The values for q, r, and the resulting fractal dimension D are given for each curve (following the computation in Sect. 5.3.

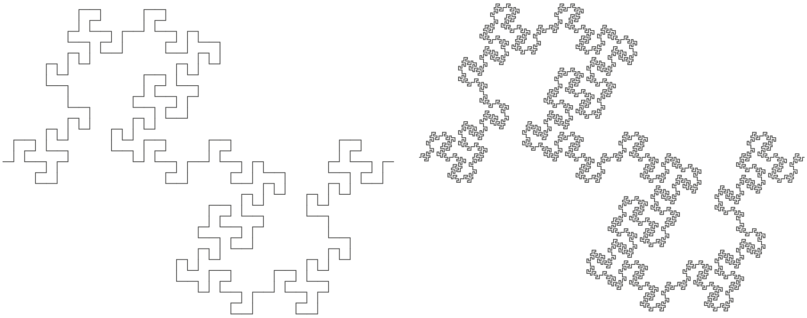
$r = 3, q = 5 \Rightarrow D \approx 1.465$:



$r = 4, q = 8 \Rightarrow D = 1.5$:



$r = 6, q = 18 \Rightarrow D \approx 1.613$:



$r = 2\sqrt{2}, q = 4 \Rightarrow D = \frac{4}{3}$:

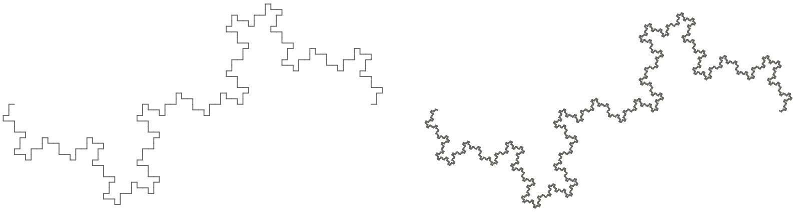


Fig. A.5 Fractal curves resulting from the generators given in Exercise 5.7

A.6 Sierpinski Curves

6.3 The construction of a turtle-based grammar for the Sierpinski curve is discussed in Sect. 14.2 – see Fig. 14.8, in particular.

6.4 Exercise 6.1 leads to a grammar with eight non-terminals, which correspond to eight congruency classes of subtriangles for such generalised Sierpinski curves. The proof for congruency of the patterns in Sect. 6.2.2 has to be extended to the remaining four congruency classes (but works in exactly the same way).

A.7 Further Space-Filling Curves

7.1 Our standard arithmetisation technique, applied to Morton order, leads to the following equation for the Morton mapping $m(t)$:

$$m(0_4.q_1q_2q_3\ldots) = \lim_{n\to\infty} M_{q_1} \circ M_{q_2} \circ \cdots \circ M_{q_n} \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \quad \text{with}$$

$$M_i \begin{pmatrix} x \\ y \end{pmatrix} = \frac{1}{2}\left[\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} + b_i\right] = \frac{1}{2}\left[\begin{pmatrix} x \\ y \end{pmatrix} + b_i\right].$$

The components of the translation vector b_i are both either 0 or 1. Applying the same technique as in Sect. 4.6.2, we obtain

$$m(0_4.q_1q_2q_3\ldots) = \frac{1}{2}b_{q_1} + \frac{1}{2^2}b_{q_2} + \frac{1}{2^3}b_{q_3} + \ldots,$$

which corresponds to a binary representation.

7.3 From Fig. A.6, we can derive the following grammar to describe the approximating polygons of the Gosper curve:

$$\begin{array}{rcl} G & \longleftarrow & G\ l\ R\ l\ R\ r\ G\ r\ G\ l\ G\ r\ R \\ & | & \uparrow l \uparrow ll \uparrow r \uparrow rr \uparrow\uparrow r \uparrow \\ R & \longleftarrow & G\ l\ R\ r\ R\ l\ R\ l\ G\ r\ G\ r\ R \\ & | & \uparrow l \uparrow\uparrow ll \uparrow l \uparrow rr \uparrow r \uparrow \end{array}$$

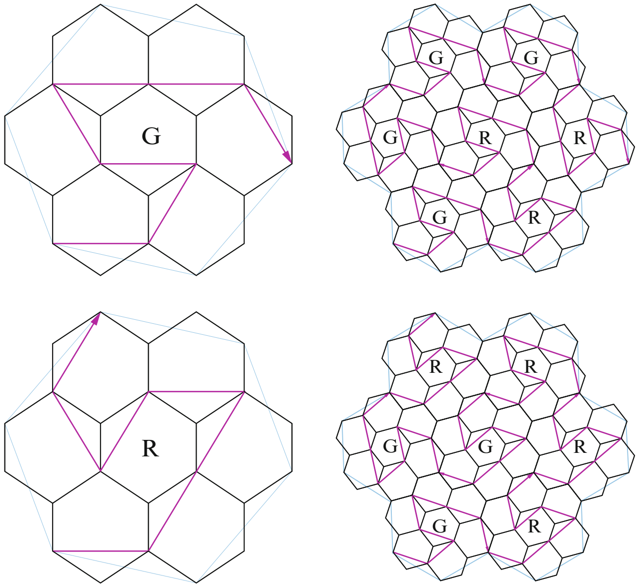


Fig. A.6 The first two approximating polygons of the Gosper curve. Again, *G* and *R* represent the two basic generating patterns

A.8 Space-Filling Curves in 3D

8.1 The operators for the approximation of Sagan's 3D Hilbert curve [233] are:

$$H_0\begin{pmatrix}x\\y\\z\end{pmatrix}=\begin{pmatrix}\frac{1}{2}x+0\\\frac{1}{2}z+0\\\frac{1}{2}y+0\end{pmatrix}\qquad H_1\begin{pmatrix}x\\y\\z\end{pmatrix}=\begin{pmatrix}\frac{1}{2}z+0\\\frac{1}{2}y+\frac{1}{2}\\\frac{1}{2}x+0\end{pmatrix}$$

$$H_2\begin{pmatrix}x\\y\\z\end{pmatrix}=\begin{pmatrix}\frac{1}{2}x+\frac{1}{2}\\\frac{1}{2}y+\frac{1}{2}\\\frac{1}{2}z+0\end{pmatrix}\qquad H_3\begin{pmatrix}x\\y\\z\end{pmatrix}=\begin{pmatrix}\frac{1}{2}z+\frac{1}{2}\\-\frac{1}{2}x+\frac{1}{2}\\-\frac{1}{2}y+\frac{1}{2}\end{pmatrix}$$

$$H_4\begin{pmatrix}x\\y\\z\end{pmatrix}=\begin{pmatrix}-\frac{1}{2}z+1\\-\frac{1}{2}x+\frac{1}{2}\\\frac{1}{2}y+\frac{1}{2}\end{pmatrix}\qquad H_5\begin{pmatrix}x\\y\\z\end{pmatrix}=\begin{pmatrix}\frac{1}{2}x+\frac{1}{2}\\\frac{1}{2}y+\frac{1}{2}\\\frac{1}{2}z+\frac{1}{2}\end{pmatrix}$$

$$H_6\begin{pmatrix}x\\y\\z\end{pmatrix}=\begin{pmatrix}-\frac{1}{2}z+\frac{1}{2}\\\frac{1}{2}y+\frac{1}{2}\\-\frac{1}{2}x+1\end{pmatrix}\qquad H_7\begin{pmatrix}x\\y\\z\end{pmatrix}=\begin{pmatrix}\frac{1}{2}x+0\\-\frac{1}{2}z+\frac{1}{2}\\-\frac{1}{2}y+1\end{pmatrix}$$

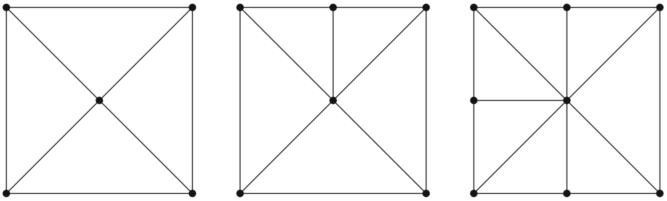


Fig. A.7 Different scenarios of constructing a conforming triangular grid in restricted quadtree cells. The *nodes* indicate the vertices of the restructed quadtree grid – *nodes on edges* indicate hanging nodes in the restricted quadtree

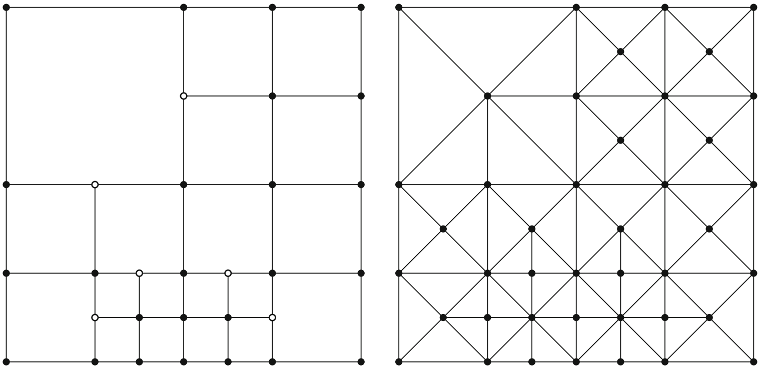


Fig. A.8 A restricted quadtree grid and its triangular counterpart (newest vertex bisection)

A.9 Refinement Trees and Space-Filling Curves

9.1 In a restricted quadtree, grid vertices can either be placed on the corners of the cells or also on the midpoints of a cell edge (if the neighbouring cell is refined). To construct a conforming grid of triangles, we replace each square cell by a set of triangle cells that cover the square cell and use all vertices – as illustrated in Fig. A.7. An example of a small quadtree grid and the corresponding triangular grid, which is compatible with newest vertex bisection, is given in Fig. A.8.

9.3 Figure A.9 shows a simple triangular grid together with the Sierpinski order on the grid cells. The Sierpinski order defines a triangle strip, i.e., a sequence of edge-connected triangle cells. When reading the vertex data A to O, we have to read one additional vertex per grid cell (in Fig. A.9, the vertices are labelled such that the data is read in alphabetical order), while two vertices can always be reused:

- In the optimal case, the two reused vertices are the two predecessors in the node stream: In our example, the first five vertices are read as ABCDE, and correspond

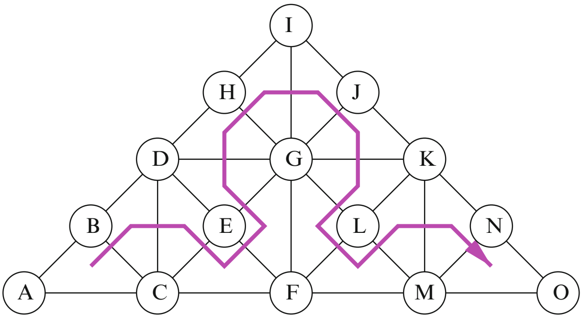


Fig. A.9 Using Sierpinski orders as triangle strips in Exercise 9.3

to triangles ABC, BCD, and CDE (i.e., the last three vertices in the stream determine the triangle).

- When reading F, however, the last two vertices were D and E, whereas the next triangle is CEF.
- One option is to swap C and D on the vertex stream. With such a swap command that exchanges the second- and third-latest vertex on the stream, our triangle strip for Fig. A.9 reads (with s as swap command):

  ABCDE s FG s **D**H s I s J s K s L s **F**M s **K**NO.

  Note that vertices D, F, and K have to be included twice in the data stream.
- Another option is to replicate all "missing" vertices within the triangle strip and thus introduce additional, duplicate triangles: Hence, after the strip ABCDE, we would need to read C again, which leads to the strip ABCDECF, in which the triangle CDE occurs twice (as CDE and DEC). The entire strip then reads:

  ABCDE**C**F**E**GDH**G**IJ**G**KL**G**F**L**MKN**M**O.

- To avoid the duplication of triangles (and respective duplicate processing), we can also allow "degenerate" triangles (where two of the vertices are identical) in the strip: Changing ABCDE to ABCDCE introduces such a degenerate triangle CDC, but now has the correct sequence CE at the end to proceed with reading F from the strip to obtain triangle CEF, The entire strip for Fig. A.9 then reads:

  ABCD**C**EF**E**GD**G**H**G**I**G**J**G**K**G**LF**L**MK**M**NO.

A.10 Parallelisation with Space-Filling Curves

10.3 Algorithm A.1 is an example on how to determine the process-local partition in a size-encoded quadtree. To keep this prototypical implementation simple, the algorithm just marks the subtrees as being local or remote. Once a subtree is entirely

Algorithm A.1: Mark partitions as local or remote in a size-encoded quadtree

```
Procedure markPart(currIndex)
    Parameter: currIndex: quadtree nodes that have already been marked (0 on entry)
    Data: sizestream: size encoding of spacetree;
          streamptr: current position
          startPartition, endPartition: interval boundaries of the local partition
    Variable: ref: size (sizestream elements) of the children (as array)
begin
    // read info on all childs from sizestream
    for i = 1, ..., 4 do
        streamptr := streamptr + 1;
        ref[i] := sizestream[streamptr];
    end
    for i = 1, ..., 4 do
        if currIndex > endPartition or currIndex+ref[i] < startPartition then
            // mark partition as remote
            markRemote(sizestream,currIndex);
            // skip partition in bitstream
            streamptr := streamptr + ref[i];
        else if currIndex ≥ startPartition and currIndex+ref[i] ≤ endPartition then
            // mark partition as local
            markLocal(sizestream,currIndex);
            // skip partition in bitstream
            streamptr := streamptr + ref[i];
        else if ref[i] > 0 then
            // recursive call to subtree (contains local and remote notes)
            markPart(currIndex);
        end
        // update variable currIndex
        currIndex = currIndex+numNodes(sizestream,streamptr);
    end
end
```

inside (or outside) the partition interval, the entire subtree is marked as local (or remote). Function `numNodes()` returns the number of nodes in a subtree – if all nodes (including inner nodes) of the tree are counted, this information can directly be obtained from the size-encoding; if only the leaf-nodes (i.e., quadtree cells) are counted, we require an additional algorithm to determine this number (and we might want to augment the size-encoding by this data). Function `numNodes()` is used to update variable *currIndex*, which holds the number of nodes (leaves only or including inner nodes) that have already been marked during the traversal.

Algorithm A.1 is a sequential algorithm, but can be modified to work in a parallel setting as illustrated in Fig. 10.5. Here, the situation might occur that a subtree that is supposed to be local is not yet stored locally – for example, during the repartitioning of a grid. Hence, Algorithm A.1 needs to be extended by respective communication operations that obtain this part from another process. Similarly, formerly local subtrees (stored as full subtrees) might be declared remote, such that the subtree representation will have to be send to the respective process.

10.5 The information required to determine left and right nodes is provided by a turtle grammar, as introduced in Sect. 3.4. See Chap. 14 for an extensive discussion.

A.11 Locality Properties of Space-Filling Curves

11.2 The following table lists the diameter-to-volume ratios for some simple geometrical objects in 2D and 3D – the last column denotes the constant c in the ratio $d = c \cdot \sqrt[D]{V}$:

Object	Typ. length	Diameter d	Area/volume V	Ratio	$c =$
Square	a	$a\sqrt{2}$	a^2	$d = \sqrt{2} \cdot V^{1/2}$	1.41
Rectangle (3:1)	$a, 3a$	$a\sqrt{10}$	$3a^2$	$d = \frac{\sqrt{10}}{\sqrt{3}} \cdot V^{1/2}$	1.83
Circle	r	$2r$	πr^2	$d = \frac{2}{\sqrt{\pi}} \cdot V^{1/2}$	1.13
Cube	a	$a\sqrt{3}$	a^3	$d = \sqrt{3} \cdot V^{1/3}$	1.73
Cuboid (3:1:1)	$a, 3a$	$a\sqrt{11}$	$3a^3$	$d = \frac{\sqrt{11}}{\sqrt[3]{3}} \cdot V^{1/3}$	2.30
Sphere	r	$2r$	$\frac{4}{3}\pi r^3$	$d = \frac{2}{\sqrt[3]{\frac{4}{3}\pi}} \cdot V^{1/3}$	1.24

A.12 Sierpinski Curves on Triangular and Tetrahedral Meshes

12.2 The 3D Sierpinski curve, as given in Sect. 8.3, is face-connected, such that a first component of the proof for Hölder continuity is in place: two parameters that are in adjacent intervals will be mapped to adjacent tetrahedral cells. An upper bound for the points' distance is thus the sum of the largest side lengths of the tetrahedra. In the standard proofs for Hölder continuity, this is put in relation with the size of the corresponding parameter intervals – in the ideal case, we have a ratio of $2^{-n} : 8^{-n}$, which for the 3D Hilbert curve means that bisecting the side length in each of the three dimension will finally lead to eight subcubes and eight corresponding subintervals.

For the face-connected 3D Sierpinski curve, this ratio is less favourable: three bisection levels are not sufficient to halve the size of each subtriangle (in the terms of maximal side length) – instead we require at least five bisection steps to guarantee this. Hence, the ratio of tetrahedral side lengths to interval sizes is more

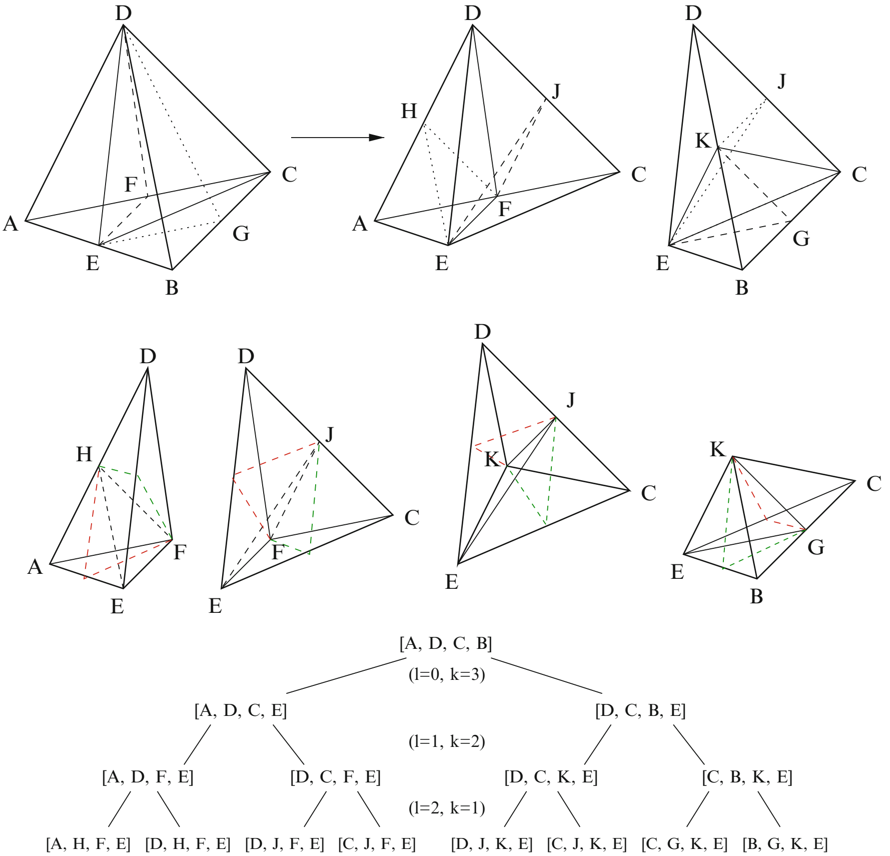


Fig. A.10 Tetrahedral refinement (generated tetrahedra and 4-tuple notation) according to the refinement rules given in Eq. (12.6) on page 192

like $2^{-n}\colon 32^{-n} = (2^5)^{-n}$. Hence, the exponent $\frac{1}{3}$ for Hölder continuity cannot be achieved – though a somewhat smaller exponent is possible.

12.3 We have already answered this question. If we stick to a uniform bisection rule, as in Sect. 8.3, we are always faced with the bottom-most situation in Fig. 12.6. Hence, we only get black tetrahedra.

12.5 Figure A.10 illustrates the first three bisection steps of a tetrahedral cell according to the bisection scheme by Maubach – see Eq. (12.6) on page 192. The starting level and sequence of nodes in the tuple notation for the initial tetrahedron were chosen to exactly match the refinement via the Baensch-Kossaczky scheme, as in Fig. 12.5 on page 185. Hence, the two schemes will produce the same sequence of child cells from identical initial tetrahedra.

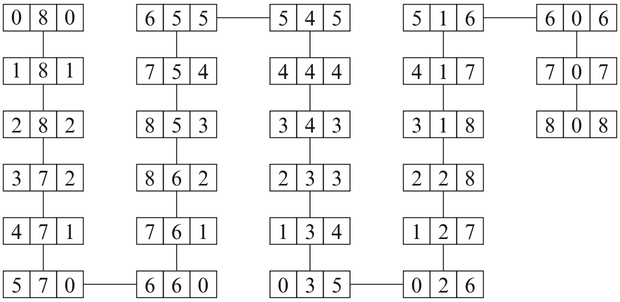


Fig. A.11 Graph representation of the operations of a 3 × 3 matrix multiplication of type $Q \mathrel{+}= QP$

A.13 Cache Efficient Algorithms for Matrix Operations

13.2 For the block operation $Q \mathrel{+}= QP$, the respective 3×3 matrix multiplication is

$$\begin{pmatrix} a_6 & a_5 & a_0 \\ a_7 & a_4 & a_1 \\ a_8 & a_3 & a_2 \end{pmatrix} \begin{pmatrix} b_0 & b_5 & b_6 \\ b_1 & b_4 & b_7 \\ b_2 & b_3 & b_8 \end{pmatrix} = \begin{pmatrix} c_6 & c_5 & c_0 \\ c_7 & c_4 & c_1 \\ c_8 & c_3 & c_2 \end{pmatrix}. \tag{A.2}$$

The derivation of the optimal execution order is illustrated in Fig. A.11. There, we connect only those operations where indices of successively accessed matrices are either identical or differ by at most 1.

A.14 Numerical Simulation on Spacetree Grids Using Space-Filling Curves

14.2 Of the three indices of the neighbour cells, only the index on the *colour edge* (i.e., edges that are not between cells with contiguous indices) is difficult to obtain. Indices on *crossed edges* are easy, as they are the increment and decrement of the current cell index.

The straightforward option to determine crossed-edge indices is to take the Sierpinski traversal of Algorithm 14.2 (to exchange refinement info between cells), and turn it into an algorithm to exchange indices, instead. If you invest an additional integer variable per cell to store the index of the crossed-edge neighbour, you obtain a data structure that allows direct access to all edge-connected cells.

Algorithm A.2 further reduces the storage requirements of this approach: only indices on *colour edges* shall be stored – for these indices, we adopt the standard

Algorithm A.2: Sierpinski traversal to propagate refinement information (for element types H_o and H_n)

```
Procedure Ho()
    Data: bitstream: bitstream representation of spacetree (streamptr: current position);
          green,red: stacks to neighbour indices;
          input,output: streams for indices of colour-edge neighbours
    Variable: currIndex: index of the current cell;
              left/rightIndex, hypoIndex: indices of the three neighbour cells
begin
    // move to next element in bitstream
    streamptr := streamptr + 1;
    if bitstream[streamptr] then
        // update local index and determine indices of crossed-edge neighbours
        hypoIndex := currIndex;
        currIndex = currIndex + 1;
        rightIndex := currIndex + 1;
        // for colour edge, obtain Index from stack
        leftIndex := green.pop();
        // write own index to colour edge output stream
        output.push(currIndex);
    else
        // recursive call to children
        Vo();
        Ko();
    end
end

// procedure Hn() is identical to Ho() up to the following lines:
Procedure Hn() begin
    // ...
    if bitstream[streamptr] then
        // ...
        leftIndex := input.pop();
        // write own index to colour edge output stream
        green.push(currIndex);
    else
        // recursive call to children ...
    end
end
```

stack&stream approach. Algorithm A.2 implements the H_o- and H_n-pattern for this idea, which have the hypotenuse and the right leg as crossed edges, and the left leg as an *old/new* colour edge. H_o and H_n only differ in the accesses to the colour stack, so the procedure for H_n only shows the two changed statements.

14.5 Ensuring a 2:1 size balance between adjacent elements of a quadtree or octree grid can also be implemented via respective traversals, as in Algorithm 14.2.

However, we now have to synchronise the refinement status of four edges (for quadtrees) or six faces (for octrees), respectively. Also, a stack-based scheme to exchange the refinement data will not work (compare Sect. 14.3). An interesting variant is the question whether the 2:1 size balance should also be enforced between elements that are only node- or edge-connected (the latter in 3D).

References

1. D. J. Abel and D. M. Mark. A comparative analysis of some two-dimensional orderings. *International Journal of Geographical Information Systems*, 4(1):21–31, 1990.
2. D. J. Abel and J. L. Smith. A data structure and algorithm based on a linear key for a rectangle retrieval problem. *Computer Vision, Graphics, and Image Processing*, 24:1–13, 1983.
3. K. Abend, T. J. Harley, and L. N. Kanal. Classification of binary random patterns. *IEEE Transactions on Information Theory*, IT-11(4):538–544, 1965.
4. M. Aftosmis, M. Berger, and J. Melton. Robust and efficient Cartesian mesh generation for component-based geometry. In *35th Aerospace Sciences Meeting and Exhibit*, 1997. AIAA 1997-0196.
5. M. J. Aftosmis, M. J. Berger, and S. M. Murman. Applications of space-filling curves to Cartesian methods for CFD. AIAA Paper 2004-1232, 2004.
6. J. Akiyama, H. Fukuda, H. Ito, and G. Nakamura. Infinite series of generalized Gosper space filling curves. In *CJCDGCGT 2005*, volume 4381 of *Lecture Notes in Computer Science*, pages 1–9, 2007.
7. I. Al-Furaih and S. Ranka. Memory hierarchy management for iterative graph structures. In *Parallel Processing Symposium, IPPS/SPDP 1998*, pages 298–302. IEEE Computer Society, 1998.
8. F. Alauzet and A. Loseille. On the use of space filling curves for parallel anisotropic mesh adaptation. In *Proceedings of the 18th International Meshing Roundtable*, pages 337–357. Springer, 2009.
9. J. Alber and R. Niedermeier. On multidimensional curves with Hilbert property. *Theory of Computing Systems*, 33:295–312, 2000.
10. S. Aluru and F. E. Sevilgen. Parallel domain decomposition and load balancing using space-filling curves. In *Proceedings of the Fourth International Conference on High-Performance Computing*, pages 230–235. IEEE Computer Society, 1997.
11. N. Amenta, S. Choi, and G. Rote. Incremental constructions con BRIO. In *Proceedings of the Nineteenth ACM Symposium on Computational Geometry*, pages 211–219, 2003.
12. M. Amor, F. Arguello, J. López, O. Plata, and E. L. Zapata. A data-parallel formulation for divide and conquer algorithms. *The Computer Journal*, 44(4):303–320, 2001.
13. E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. DuCroz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. LAPACK: a portable linear algebra library for high-performance computers. Technical Report CS-90-105, LAPACK Working Note #20, University of Tennessee, Knoxville, TN, 1990.
14. J. A. Anderson, C. D. Lorenz, and A. Travesset. General purpose molecular dynamics simulations fully implemented on graphics processing units. *Journal of Computational Physics*, 227(10):5342–5359, 2008.

M. Bader, *Space-Filling Curves*, Texts in Computational Science and Engineering 9,
DOI 10.1007/978-3-642-31046-1, © Springer-Verlag Berlin Heidelberg 2013

15. A. Ansari and A. Fineberg. Image data ordering and compression using Peano scan and LOT. *IEEE Transactions on Consumer Electronics*, 38(3):436–445, 1992.
16. L. Arge, M. De Berg, H. Haverkort, and K. Yi. The priority R-tree: A practically efficient and worst-case optimal R-tree. *ACM Transactions on Algorithms*, 4(1):9:1–9:29, 2008.
17. D. N. Arnold, A. Mukherjee, and L. Pouly. Locally adapted tetrahedral meshes using bisection. *SIAM Journal on Scientific Computing*, 22(2):431–448, 2000.
18. T. Asano, D. Ranjan, T. Roos, E. Welzl, and P. Widmayer. Space-filling curves and their use in the design of geometric data structures. *Theoretical Computer Science*, 181(1):3–15, 1997.
19. M. Bader. Exploiting the locality properties of Peano curves for parallel matrix multiplication. In *Proceedings of the Euro-Par 2008*, volume 5168 of *Lecture Notes in Computer Science*, pages 801–810, 2008.
20. M. Bader, C. Böck, J. Schwaiger, and C. Vigh. Dynamically adaptive simulations with minimal memory requirement – solving the shallow water equations using Sierpinksi curves. *SIAM Journal on Scientific Computing*, 32(1):212–228, 2010.
21. M. Bader, R. Franz, S. Guenther, and A. Heinecke. Hardware-oriented implementation of cache oblivious matrix operations based on space-filling curves. In *Parallel Processing and Applied Mathematics, 7th International Conference, PPAM 2007*, volume 4967 of *Lecture Notes in Computer Science*, pages 628–638, 2008.
22. M. Bader and A. Heinecke. Cache oblivious dense and sparse matrix multiplication based on Peano curves. In *Proceedings of the PARA '08, 9th International Workshop on State-of-the-Art in Scientific and Parallel Computing*, volume 6126/6127 of *Lecture Notes in Computer Science*, 2010. In print.
23. M. Bader, S. Schraufstetter, C. A Vigh, and J. Behrens. Memory efficient adaptive mesh generation and implementation of multigrid algorithms using Sierpinski curves. *International Journal of Computational Science and Engineering*, 4(1):12–21, 2008.
24. M. Bader and C. Zenger. Cache oblivious matrix multiplication using an element ordering based on a Peano curve. *Linear Algebra and Its Applications*, 417(2–3):301–313, 2006.
25. M. Bader and C. Zenger. Efficient storage and processing of adaptive triangular grids using Sierpinski curves. In *Computational Science – ICCS 2006*, volume 3991 of *Lecture Notes in Computer Science*, pages 673–680, 2006.
26. Y. Bandou and S.-I. Kamata. An address generator for an n-dimensional pseudo-Hilbert scan in a hyper-rectangular parallelepiped region. In *International Conference on Image Processing, ICIP 2000*, pages 707–714, 2000.
27. R. Bar-Yehuda and C. Gotsman. Time/space tradeoffs for polygon mesh rendering. *ACM Transactions on Graphics*, 15(2):141–152, 1996.
28. J. Barnes and P. Hut. A hierarchical $O(N \log N)$ force-calculation algorithm. *Nature*, 324:446–449, 1986.
29. J.J. Bartholdi III and P. Goldsman. Vertex-labeling algorithms for the Hilbert spacefilling curve. *Software: Practice and Experience*, 31(5):395–408, 2001.
30. J. J. Bartholdi III and P. Goldsman. Multiresolution indexing of triangulated irregular networks. *IEEE Transactions on Visualization and Computer Graphics*, 10(3):1–12, 2004.
31. J. J. Bartholdi III and L. K. Platzman. An $O(N \log N)$ planar travelling salesman heuristic based on spacefilling curves. *Operations Research Letters*, 1(4):121–125, 1982.
32. J. J. Bartholdi III and L. K. Platzman. Heuristics based on spacefilling curves for combinatorial problems in the Euclidian space. *Management Science*, 34(3):291–305, 1988.
33. A. C. Bauer and A. K. Patra. Robust and efficient domain decomposition preconditioners for adaptive hp finite element approximations of linear elasticity with and without discontinuous coefficients. *International Journal for Numerical Methods in Engineering*, 59:337–364, 2004.
34. K.E. Bauman. The dilation factor of the Peano–Hilbert curve. *Mathematical Notes*, 80(5):609–620, 2006. Translated from Matematicheskie Zametki, vol. 80, no. 5, 2006, pp. 643–656. Original Russian Text Copyright 2006 by K. E. Bauman.
35. R. Bayer. The universal B-tree for multidimensional indexing. Technical Report TUM-I9637, Institut für Informatik, Technische Universität München, 1996.

36. J. Behrens. Multilevel optimization by space-filling curves in adaptive atmospheric modeling. In F. Hülsemann, M. Kowarschik, and U. Rüde, editors, *Frontiers in Simulation - 18th Symposium on Simulation Techniques*, pages 186–196. SCS Publishing House, Erlangen, 2005.
37. J. Behrens. *Adaptive atmospheric modeling: key techniques in grid generation, data structures, and numerical operations with applications*, volume 54 of *Lecture Notes in Computational Science and Engineering*. Springer, 2006.
38. J. Behrens and M. Bader. Efficiency considerations in triangular adaptive mesh refinement. *Philosophical Transactions of the Royal Society A*, 367:4577–4589, 2009. Theme Issue 'Mesh generation and mesh adaptation for large-scale Earth-system modelling'.
39. J. Behrens, N. Rakowsky, W. Hiller, D. Handorf, M. Läuter, J. Päpke, and K. Dethloff. Parallel adaptive mesh generator for atmospheric and oceanic simulation. *Ocean Modelling*, 10:171–183, 2005.
40. J. Behrens and J. Zimmermann. Parallelizing an unstructured grid generator with a space-filling curve approach. In *Euro-Par 2000 Parallel Processing*, volume 1900 of *Lecture Notes in Computer Science*, pages 815–823, 2000.
41. D. Bertsimas and M. Grigni. Worst-case example for the spacefilling curve heuristics for the Euclidian traveling salesman problem. *Operations Research Letters*, 8:241–244, 1989.
42. T. Bially. Space-filling curves: Their generation and their application to bandwidth reduction. *IEEE Transactions on Information Theory*, IT-15(6):658–664, 1969.
43. E. Bänsch. Local mesh refinement in 2 and 3 dimensions. *IMPACT of Computing in Science and Engineering*, 3(3):181–191, 1991.
44. A. Bogomjakov and C. Gotsman. Universal rendering sequences for transparent vertex caching of progressive meshes. *Computer Graphics forum*, 21(2):137–148, 2002.
45. E. Borel. *Elements de la Theorie des Ensembles.* Editions Albin Michel, Paris, 1949. Note IV: La courbe de Péano.
46. V. Brázdová and D. R. Bowler. Automatic data distribution and load balancing with space-filling curves: implementation in CONQUEST. *Journal of Physics: Condensed Matter*, 20, 2008.
47. G. Breinholt and Ch. Schierz. Algorithm 781: Generating Hilbert's space-filling curve by recursion. *ACM Transactions on Mathematical Software*, 24(2):184–189, 1998.
48. M. Brenk, H.-J. Bungartz, M. Mehl, I. L. Muntean, T. Neckel, and T. Weinzierl. Numerical simulation of particle transport in a drift ratchet. *SIAM Journal on Scientific Computing*, 30(6):2777–2798, 2008.
49. K. Brix, S. Sorana Melian, S. Müller, and G. Schieffer. Parallelisation of multiscale-based grid adaption using space-filling curves. *ESAIM: Proceedings*, 29:108–129, 2009.
50. K. Buchin. Constructing Delauney triangulations along space-filling curves. In *ESA 2009*, volume 5757, pages 119–130, 2009.
51. E. Bugnion, T. Roos, R. Wattenhofer, and P. Widmayer. Space filling curves versus random walks. In *Algorithmic Foundations of Geographic Information Systems*, volume 1340, pages 199–211, 1997.
52. H.-J. Bungartz, M. Mehl, T. Neckel, and T. Weinzierl. The PDE framework Peano applied to fluid dynamics: an efficient implementation of a parallel multiscale fluid dynamics solver on octree-like adaptive Cartesian grids. *Computational Mechanics*, 46(1):103–114, 2010.
53. H.-J. Bungartz, M. Mehl, and T. Weinzierl. A parallel adaptive Cartesian PDE solver using space-filling curves. In E.W. Nagel, V.W. Walter, and W. Lehner, editors, *Euro-Par 2006, Parallel Processing, 12th International Euro-Par Conference*, volume 4128 of *Lecture Notes in Computer Science*, pages 1064–1074, 2006.
54. C. Burstedde, O. Ghattas, M. Gurnis, G. Stadler, E. Tan, T. Tu, L. C. Wilcox, and S. Zhong. Scalable adaptive mantle convection simulation on petascale supercomputers. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pages 1–15. IEEE Press, 2008.
55. C. Burstedde, L. C. Wilcox, and O. Ghattas. p4est: Scalable algorithms for parallel adaptive mesh refinement on forests of octrees. *SIAM Journal on Scientific Computing*, 33(3):1103–1133, 2011.

56. A. Buttari, J. Dongarra, J. Kurzak, J. Langou, P. Luszczek, and S. Tomov. The impact of multicore on math software. In *Applied Parallel Computing, State of the Art in Scientific Computing*, volume 4699 of *Lecture Notes in Computer Science*, pages 1–10, 2007.
57. A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. Technical Report UT-CS-07-600, LAPACK Working Note #191, ICL, University Tennessee, 2007.
58. A. R. Butz. Convergence with Hilbert's space filling curve. *Journal of Computer and System Sciences*, 3:128–146, 1969.
59. A. R. Butz. Alternative algorithm for Hilbert's space-filling curve. *IEEE Transactions on Computers*, C-20(4):424–426, 1971.
60. A. C. Calder, B. C. Curtis, L. J. Dursi, B. Fryxell, P. MacNeice, K. Olson, P. Ricker, R. Rosner, F. X. Timmes, H. M. Tufo, J. W. Turan, M. Zingale, and G. Henry. High performance reactive fluid flow simulations using adaptive mesh refinement on thousands of processors. In *Supercomputing '00: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, page 56. IEEE Computer Society, 2000.
61. P. M. Campbell, K. D. Devine, J. E. Flaherty, L. G. Gervasio, and J. D. Terescoy. Dynamic octree load balancing using space-filling curves. Technical Report CS-03-01, Williams College Department of Computer Science, 2003.
62. X. Cao and Z. Mo. A new scalable parallel method for molecular dynamics based on cell-block data structure. In *Parallel and Distributed Processing and Applications*, 3358, pages 757–764, 2005.
63. J. Castro, M. Georgiopoulos, R. Demara, and A. Gonzalez. Data-partitioning using the hilbert space filling curves: Effect on the speed of convergence of Fuzzy ARTMAP for large database problems. *Neural Networks*, 18:967–984, 2005.
64. E. Cesaro. Remarques sur la courbe de von Koch. *Atti della R. Accad. della Scienze fisiche e matem. Napoli*, 12(15):1–12, 1905. Reprinted in: Opere scelte, a cura dell'Unione matematica italiana e col contributo del Consiglio nazionale delle ricerche, Vol. 2: Geometria, analisi, fisica matematica. Rome: Edizioni Cremonese, pp. 464-479, 1964.
65. S. Chatterjee, V. V. Jain, A. R. Lebeck, S. Mundhra, and M. Thottethodi. Nonlinear array layouts for hierarchical memory systems. In *International Conference on Supercomputing (ICS'99).* ACM, New York, 1999.
66. H.-L. Chen and Y.-I. Chang. Neighbor-finding based on space-filling curves. *Information Systems*, 30(3):205–226, 2005.
67. G. Chochia, M. Cole, and T. Heywood. Implementing the hierarchical PRAM on the 2d mesh: analyses and experiments. *IEEE Symposium on Parallel and Distributed Processing*, 0:587–595, 1995. Preprint on http://homepages.inf.ed.ac.uk/mic/Pubs/ECS-CSG-10-95.ps.gz.
68. W. J. Coirier and K. G. Powell. Solution-adaptive Cartesian cell approach for viscous and inviscid fluid flows. *AIAA Journal*, 34(5):938–945, 1996.
69. A. J. Cole. A note on space-filling curves. *Software: Practice and Experience*, 13:1181–1189, 1983.
70. S. Craver, B.-L. Yeo, and M. Yeung. Multilinearization data structure for image browsing. In *Storage and Retrieval for Image and Video Databases VII*, volume 3656, pages 155–166, 1998.
71. R. Dafner, D. Cohen-Or, and Y. Matias. Context-based space filling curves. *Computer Graphics Forum*, 19(3):209–218, 2000.
72. K. Daubner. Geometrische Modellierung mittels Oktalbäumen und Visualisierung von Simulationsdaten aus der Strömungsmechanik. Institut für Informatik, Technische Universität München, 2005.
73. L. De Floriani and E. Puppo. Hierarchical triangulation for multiresolution surface description. *ACM Transactions on Graphics*, 14(4):363–411, 1995.
74. J. M. Dennis. Partitioning with space-filling curves on the cubed-sphere. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, pages 269–275. IEEE Computer Society, 2003.

75. J. M. Dennis. Inverse space-filling curve partitioning of a global ocean model. In *Parallel and Distributed Processing Symposium, IPDPS 2007*, pages 1–10. IEEE International, 2007.
76. K. D. Devine, E. G. Boman, R. T. Heaphy, B. A. Hendrickson, J. D. Terescoy, J. Falk, J. E. Flaherty, and L. G. Gervasio. New challenges in dynamic load balancing. *Applied Numerical Mathematics*, 52:133–152, 2005.
77. R. Diekmann, J. Hungershöfer, M. Lux, M. Taenzer, and J.-M. Wierum. Using space filling curves for efficient contact searching. In *16th IMACS World Congress*, 2000.
78. J. J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–28, 1990.
79. J. Dreher and R. Grauer. Racoon: A parallel mesh-adaptive framework for hyperbolic conservation laws. *Parallel Computing*, 31(8–9):913–932, 2005.
80. M. Duchaineau, M. Wolinsky, D. E. Sigeti, M. C. Miller, C. Aldrich, and M. B. Mineev-Weinstein. ROAMing terrain: real-time optimally adapting meshes. In *VIS '97: Proceedings of the 8th Conference on Visualization '97*, pages 81–88. IEEE Computer Society Press, 1997.
81. E. Elmroth, F. Gustavson, I. Jonsson, and B. Kågström. Recursive blocked algorithms and hybrid data structures for dense matrix library software. *SIAM Review*, 46(1):3–45, 2004.
82. M. Elshafei and M. S. Ahmed. Fuzzification using space-filling curves. *Intelligent Automation and Soft Computing*, 7(2):145–157, 2001.
83. M. Elshafei-Ahmed. Fast methods for split codebooks. *Signal Processing*, 80:2553–2565, 2000.
84. W. Evans, D. Kirkpatrick, and G. Townsend. Right-triangulated irregular networks. *Algorithmica*, 30(2):264–286, 2001.
85. C. Faloutsos. Analytical results on the quadtree decomposition of arbitrary rectangles. *Pattern Recognition Letters*, 13:31–40, 1992.
86. C. Faloutsos and S. Roseman. Fractals for secondary key retrieval. In *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 247–252, 1989.
87. R. Finkel and Bentley J. L. Quad trees: a data structure for retrieval on composite keys. *Acta Informatica*, 4(1):1–9, 1974.
88. J. E. Flaherty, R. M. Loy, M. S. Shephard, B. K. Szymanski, J. D. Terescoy, and L. H. Ziantz. Adaptive local refinement with octree load balancing for the parallel solution of three-dimensional conservation laws. *Journal of Parallel and Distributed Computing*, 47:139–152, 1997.
89. A. C. Frank. *Organisationsprinzipien zur Integration von geometrischer Modellierung, numerischer Simulation und Visualisierung*. Herbert Utz Verlag, Dissertation, Institut für Informatik, Technische Universität München, 2000.
90. J. Frens and D. S. Wise. Auto-blocking matrix-multiplication or tracking BLAS3 performance from source code. In *Proceedings of the 6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 206–216, 1997.
91. J. Frens and D. S. Wise. QR factorization with Morton-ordered quadtree matrices for memory re-use and parallelism. In *Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 144–154, 2003.
92. M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, pages 285–297, 1999.
93. H. Fukuda, M. Shimizu, and G. Nakamura. New Gosper space filling curves. In *Proceedings of the International Conference on Computer Graphics and Imaging (CGIM2001)*, pages 34–38, 2001.
94. J. Gao and J. M. Steele. General spacefilling curve heuristics and limit theory for the traveling salesman problem. *Journal of Complexity*, 10:230–245, 1994.
95. M. Gardner. Mathematical games – in which "monster" curves force redefinition of the word "curve". *Scientific American*, 235:124–133, Dec. 1976.
96. I. Gargantini. An effective way to represent quadtrees. *Communications of the ACM*, 25(12):905–910, 1982.

97. T. Gerstner. Multiresolution Compression and Visualization of Global Topographic Data. *GeoInformatica*, 7(1):7–32, 2003. (shortened version in Proc. Spatial Data Handling 2000, P. Forer, A.G.O. Yeh, J. He (eds.), pp. 14–27, IGU/GISc, 2000, also as SFB 256 report 29, Univ. Bonn, 1999).
98. P. Gibbon, W. Frings, S. Dominiczak, and B. Mohr. Performance analysis and visualization of the n-body tree code PEPC on massively parallel computers. In *Parallel Computing: Current & Future Issues of High-End Computing, Proceedings of the International Conference ParCo 2005*, volume 33 of *NIC Series*, pages 367–374, 2006.
99. W. Gilbert. A cube-filling Hilbert curve. *The Mathematical Intelligencer*, 6(3):78–79, 1984.
100. J. Gips. *Shape Grammars and their Uses*. Interdisciplinary Systems Research. Birkhäuser Verlag, 1975.
101. L. M. Goldschlager. Short algorithms for space-filling curves. *Software: Practice and Experience*, 11:99–100, 1981.
102. M. F. Goodchild and D. M. Mark. The fractal nature of geographic phenomena. *Annals of the Association of American Geographers*, 77(2):265–278, 1987.
103. K. Goto and R. van de Geijn. On reducing TLB misses in matrix multiplication. FLAME working note #9. Technical Report TR-2002-55, The University of Texas at Austin, Department of Computer Sciences, 2002.
104. K. Goto and R. A. van de Geijn. Anatomy of a high-performance matrix multiplication. *ACM Transactions on Mathematical Software*, 34(3):12:1–12:25, 2008.
105. C. Gotsman and M. Lindenbaum. On the metric properties of space-filling curves. *IEEE Transactions on Image Processing*, 5(5):794–797, 1996.
106. P. Gottschling, D. S. Wise, and A. Joshi. Generic support of algorithmic and structural recursion for scientific computing. *International Journal of Parallel, Emergent and Distributed Systems*, 24(6):479–503, 2009.
107. M. Griebel and M. A. Schweitzer. A particle-partition of unity method—part IV: Parallelization. In *Meshfree Methods for Partial Differential Equations*, volume 26 of *Lecture Notes in Computational Science and Engineering*, pages 161–192, 2002.
108. M. Griebel and G. Zumbusch. Parallel multigrid in an adaptive PDE solver based on hashing and space-filling curves. *Parallel Computing*, 27(7):827–843, 1999.
109. M. Griebel and G. Zumbusch. Hash based adaptive parallel multilevel methods with space-filling curves. In H. Rollnik and D. Wolf, editors, *NIC Symposium 2001*, volume 9 of *NIC Series*, pages 479–492. Forschungszentrum Jülich, 2002.
110. J. G. Griffiths. Table-driven algorithm for generating space-filling curves. *Computer Aided Design*, 17(1):37–41, 1985.
111. J. G. Griffiths. An algorithm for displaying a class of space-filling curves. *Software: Practice and Experience*, 16(5):403–411, 1986.
112. J. Gunnels, F. Gustavson, K. Pingali, and K. Yotov. Is cache-oblivious DGEMM viable? In *Applied Parallel Computing. State of the Art in Scientific Computing*, volume 4699 of *Lecture Notes in Computer Science*, pages 919–928, 2007.
113. J. A. Gunnels, F. G. Gustavson, G. M. Henry, and R. A. van de Geijn. FLAME: formal linear algebra methods environment. *ACM Transactions on Mathematical Software*, 27(4):422–455, 2001.
114. F. Günther, M. Mehl, M. Pögl, and C. Zenger. A cache-aware algorithm for PDEs on hierarchical data structures based on space-filling curves. *SIAM Journal on Scientific Computing*, 28(5):1634–1650, 2006.
115. F. Gustavson, L. Karlsson, and B. Kågström. Parallel and cache-efficient in-place matrix storage format conversion. *Transactions on Mathematical Software*, 38(3):17:1–17:32, 2012.
116. F. G. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM Journal of Research and Development*, 41(6), 1999.
117. G. Haase, M. Liebmann, and G. Plank. A Hilbert-order multiplication scheme for unstructured sparse matrices. *International Journal of Parallel, Emergent and Distributed Systems*, 22(4):213–220, 2007.

118. C. H. Hamilton and A. Rau-Chaplin. Compact Hilbert indices: Space-filling curves for domains with unequal side lengths. *Information Processing Letters*, 105:155–163, 2008.
119. H. Han and C.-W. Tseng. Exploiting locality for irregular scientific codes. *IEEE Transactions on Parallel and Distributed Systems*, 17(7):606–618, 2006.
120. J. Hartmann, A. Krahnke, and C. Zenger. Cache efficient data structures and algorithms for adaptive multidimensional multilevel finite element solvers. *Applied Numerical Mathematics*, 58(4):435–448, 2008.
121. A. Haug. *Sierpinski-Kurven zur speichereffizienten numerischen Simulation auf adaptiven Tetraedergittern*. Diplomarbeit, Fakultät für Informatik, Technische Universität München, 2006.
122. H. Haverkort and F. van Walderveen. Four-dimensional Hilbert curves for R-trees. In *Proceedings of the Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 63–73, 2009.
123. H. Haverkort and F. van Walderveen. Locality and bounding-box quality of two-dimensional space-filling curves. *Computational Geometry: Theory and Applications*, 43(2):131–147, 2010.
124. G. Heber, R. Biswas, and G. R. Gao. Self-avoiding walks over adaptive unstructured grids. *Concurrency: Practice and Experience*, 12:85–109, 2000.
125. D. J. Hebert. Symbolic local refinement of tetrahedral grids. *Journal of Symbolic Computation*, 17(5):457–472, 1994.
126. D. J. Hebert. Cyclic interlaced quadtree algorithms for quincunx multiresolution. *Journal of Algorithms*, 27:97–128, 1998.
127. A. Heinecke and M. Bader. Parallel matrix multiplication based on space-filling curves on shared memory multicore platforms. In *Proceedings of the 2008 Computing Frontiers Conference and co-located workshops: MAW'08 and WRFT'08*, pages 385–392. ACM, 2008.
128. B. Hendrickson. Load balancing fictions, falsehoods and fallacies. *Applied Mathematical Modelling*, 25:99–108, 2000.
129. B. Hendrickson and K. Devine. Dynamic load balancing in computational mechanics. *Computer Methods in Applied Mechanical Engineering*, 184:485–500, 2000.
130. J. R. Herrero and J. J. Navarro. Analysis of a sparse hypermatrix Cholesky with fixed-sized blocking. *Applicable Algebra in Engineering, Communication and Computing*, 18(3):279–295, 2007.
131. D. Hilbert. Über die stetige Abbildung einer Linie auf ein Flächenstück. *Mathematische Annalen*, 38:459–460, 1891. Available online on the Göttinger Digitalisierungszentrum.
132. J.-W. Hong and H. T. Kung. I/O complexity: the red-blue pebble game. In *Proceedings of ACM Symposium on Theory of Computing*, pages 326–333, 1981.
133. H. Hoppe. Optimization of mesh locality for transparent vertex caching. In *SIGGRAPH '99: Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, pages 269–276. ACM Press/Addison-Wesley Publishing Co., 1999.
134. Y. C. Hu, A. Cox, and W. Zwaenepoel. Improving fine-grained irregular shared-memory benchmarks by data reordering. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, pages # 33. IEEE Computer Society, 2000.
135. J. Hungershöfer and J.-M. Wierum. On the quality of partitions based on space-filling curves. In *ICCS 2002*, volume 2331 of *Lecture Notes in Computer Science*, pages 36–45, 2002.
136. G. M. Hunter and K. Steiglitz. Operations on images using quad trees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-1(2):145–154, 1979.
137. L. M. Hwa, M. A. Duchaineau, and K.i. Joy. Adaptive 4-8 texture hierarchies. In *VIS '04: Proceedings of the Conference on Visualization '04*, pages 219–226. IEEE Computer Society, 2004.
138. C. Jackings and S. L. Tanimoto. Octrees and their use in representing three-dimensional objects. *Computer Graphics and Image Processing*, 14(31):249–270, 1980.
139. H. V. Jagadish. Linear clustering of objects with multiple attributes. *ACM SIGMOD Record*, 19(2):332–342, 1990.

140. H. V. Jagadish. Analysis of the Hilbert curve for representing two-dimensional space. *Information Processing Letters*, 62(1):17–22, 1997.
141. G. Jin and J. Mellor-Crummey. SFCGen: a framework for efficient generation of multi-dimensional space-filling curves by recursion. *ACM Transactions on Mathematical Software*, 31(1):120–148, 2005.
142. Bentley J. L. and D. F. Stanat. Analysis of range searches in quad trees. *Information Processing Letters*, 3(6):170–173, 1975.
143. M. Kaddoura, C.-W. Ou, and S. Ranka. Partitioning unstructured computational graphs for nonuniform and adaptive environments. *IEEE Concurrency*, 3(3):63–69, 1995.
144. C. Kaklamanis and G. Persiano. Branch-and-bound and backtrack search on mesh-connected arrays of processors. *Mathematical Systems Theory*, 27:471–489, 1994.
145. S. Kamata, R. O. Eason, and Y. Bandou. A new algorithm for n-dimensional Hilbert scanning. *IEEE Transactions on Image Processing*, 8(7):964–973, 1999.
146. I. Kamel and C. Faloutsos. On packing R-trees. In *Proceedings of the Second International ACM Conference on Information and Knowledge Management*, pages 490–499. ACM New York, 1993.
147. E. Kawaguchi and T. Endo. On a method of binary-picture representation and its application to data compression. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-2(1):27–35, 1980.
148. A. Klinger. Data structures and pattern recognition. In *Proceedings of the First International Joint Conference on Pattern Recognition*, pages 497–498. IEEE, 1973.
149. A. Klinger and C. R. Dyer. Experiments on picture representation using regular decomposition. *Computer Graphics and Image Processing*, 5:68–106, 1976.
150. K. Knopp. Einheitliche Erzeugung und Darstellung der Kurven von Peano, Osgood und von Koch. *Archiv der Mathematik und Physik*, 26:103–115, 1917.
151. I. Kossaczky. A recursive approach to local mesh refinement in two and three dimensions. *Journal of Computational and Applied Mathematics*, 55:275–288, 1994.
152. R. Kriemann. Parallel $\mathcal{H}$-matrix arithmetics on shared memory systems. *Computing*, 74:273–297, 2005.
153. J. P. Lauzon, D. M. Mark, L. Kikuchi, and J. A. Guevara. Two-dimensional run-encoding for quadtree representation. *Computer Vision, Graphics, and Image Processing*, 30(1):56–69, 1985.
154. J. K. Lawder and P. J. H. King. Querying multi-dimensional data indexed using the Hilbert space-filling curve. *ACM SIGMOD Record*, 30(1):19–24, 2001.
155. J. K. Lawder and P. J. H. King. Using state diagrams for Hilbert curve mappings. *International Journal of Computer Mathematics*, 78:327–342, 2001.
156. D. Lea. Digital and Hilbert k-d-trees. *Information Processing Letters*, 27:35–41, 1988.
157. H. L. Lebesgue. *Leçons sur l'intégration et la recherche des fonctions primitives*. Gauthier-Villars, Paris, 1904. Available online on the University of Michigan Historical Math Collection.
158. J.-H. Lee and Y.-C. Hsueh. Texture classification method using multiple space filling curves. *Pattern Recognition Letters*, 15:1241–1244, 1994.
159. M. Lee and H. Samet. Navigating through triangle meshes implemented as linear quadtrees. *ACM Transactions on Graphics*, 19(2):79–121, 2000.
160. A. Lempel and J. Ziv. Compression of two-dimensional data. *IEEE Transactions on Information Theory*, IT-32(1):2–8, 1986.
161. S. Liao, M. A. Lopez, and S. T. Leutenegger. High dimensional similarity search with space filling curves. In *Proceedings of the 17th International Conference on Data Engineering*, pages 615–622. IEEE Computer Society, 2000.
162. A. Lindenmayer. Mathematical models for cellular interactions in development. *Journal of Theoretical Biology*, 18:280–299, 1968.
163. P. Lindstrom, D. Koller, W. Ribarsky, L. F. Hodges, N. Faust, and G. A. Turner. Real-time, continuous level of detail rendering of height fields. In *SIGGRAPH '96: Proceedings of the*

23rd Annual Conference on Computer Graphics and Interactive Techniques, pages 109–118. ACM, 1996.

164. P. Lindstrom and V. Pascucci. Terrain simplification simplified: A general framework for view-dependent out-of-core visualization. Technical Report UCRL-JC-147847, 2002.
165. A Liu and B. Joe. On the shape of tetrahedra from bisection. *Mathematics of Computation*, 63:141–154, 1994.
166. A Liu and B. Joe. Quality local refinement of tetrahedral meshes based on bisection. *SIAM Journal on Scientific Computing*, 16:1269–1291, 1995.
167. P. Liu and S. N. Bhatt. Experiences with parallel n-body simulation. *IEEE Transactions on Parallel and Distributed Systems*, 11(12):1306–1323, 2000.
168. X. Liu. Four alternative patterns of the Hilbert curve. *Applied Mathematics and Communication*, 147:741–752, 2004.
169. X. Liu and G. Schrack. Encoding and decoding the Hilbert order. *Software: Practice and Experience*, 26(12):1335–1346, 1996.
170. X. Liu and G. F. Schrack. A new ordering strategy applied to spatial data processing. *International Journal of Geographical Information Science*, 12(1):3–22, 1998.
171. Y. Liu and J. Snoeyink. A comparison of five implementations of 3d Delaunay tessellation. *Combinatorial and Computational Geometry*, 52:439–458, 2005.
172. J. Luitjens, M. Berzins, and T. Henderson. Parallel space-filling curve generation through sorting. *Concurrency and Computation: Practice and Experience*, 19:1387–1402, 2007.
173. G. Mainar-Ruiz and J.-C. Perez-Cortes. Approximate nearest neighbor search using a single space-filling curve and multiple representations of the data points. In *18th International Conference on Pattern Recognition, 2006 – ICPR 2006*, pages 502–505, 2006.
174. B. Mandelbrot. *The Fractal Geometry of Nature*. Freeman and Company, 1977, 1982, 1983.
175. Y. Matias and A. Shamir. A video scrambling technique based on space filling curves. In *Advances in Cryptology – CRYPTO '87*, volume 293, pages 398–417, 1987.
176. J. M. Maubach. Local bisection refinement for n-simplicial grids generated by reflection. *SIAM Journal on Scientific Computing*, 16(1):210–227, 1995.
177. J. M. Maubach. Space-filling curves for 2-simplicial meshes created with bisections and reflections. *Applications of Mathematics*, 3:309–321, 2005.
178. D. Meagher. Geometric modelling using octree encoding. *Computer Graphics and Image Processing*, 19:129–147, 1980.
179. D. Meagher. Octree encoding: A new technique for the representation, manipulation and display of arbitrary 3d objects by computer. Technical report, Image Processing Laboratory, Rensselaer Polytechnic Institute, 1980.
180. M. Mehl, M. Brenk, H.-J. Bungartz, K. Daubner, I. L. Muntean, and T. Neckel. An Eulerian approach for partitioned fluid-structure simulations on Cartesian grids. *Computational Mechanics*, 43(1):115–124, 2008.
181. M. Mehl, T. Neckel, and P. Neumann. Navier-stokes and lattice-boltzmann on octree-like grids in the Peano framework. *International Journal for Numerical Methods in Fluids*, 65(1):67–86, 2010.
182. M. Mehl, T. Neckel, and T. Weinzierl. Concepts for the efficient implementation of domain decomposition approaches for fluid-structure interactions. In U. Langer, M. Discacciati, D.E. Keyes, O.B. Widlund, and W. Zulehner, editors, *Domain Decomposition Methods in Science and Engineering XVII*, volume 60 of *Lecture Notes in Science an Enginnering*, 2008.
183. M. Mehl, T. Weinzierl, and C. Zenger. A cache-oblivious self-adaptive full multigrid method. *Numerical Linear Algebra with Applications*, 13(2–3):275–291, 2006.
184. J. Mellor-Crummey, D. Whalley, and K. Kennedy. Improving memory hierarchy performance for irregular applications using data and computation reorderings. *International Journal of Parallel Programming*, 29(3):217–247, 2001.
185. N. Memon, D. L. Neuhoff, and S. Shende. An analysis of some common scanning techniques for lossless image coding. *IEEE Transactions on Image Processing*, 9(11):1837–1848, 2000.
186. R. Miller and Q. F. Stout. Mesh computer algorithms for computational geometry. *IEEE Transactions on Computers*, 38(3):321–340, 1989.

187. W. F. Mitchell. Adaptive refinement for arbitrary finite-element spaces with hierarchical bases. *Journal of Computational and Applied Mathematics*, 36:65–78, 1991.
188. W. F. Mitchell. Hamiltonian paths through two- and three-dimensional grids. *Journal of Research of the National Institute of Standards and Technology*, 110(2):127–136, 2005.
189. W. F. Mitchell. A refinement-tree based partitioning method for dynamic load balancing with adaptively refined grids. *Journal of Parallel and Distributed Computing*, 67(4):417–429, 2007.
190. G. Mitchison and R. Durbin. Optimal numberings of an $N \times N$-array. *SIAM Journal on Algebraic and Discrete Methods*, 7(4):571–582, 1986.
191. B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz. Analysis of the clustering properties of the Hilbert space-filling curve. *IEEE Transactions on Knowledge and Data Engineering*, 13(1):124–141, 2001.
192. A. Mooney, J. G. Keating, and D. M. Heffernan. A detailed study of the generation of optically detectable watermarks using the logistic map. *Chaos, Solitons and Fractals*, 30(5):1088–1097, 2006.
193. E. H. Moore. On certain crinkly curves. *Transactions of the American Mathematical Society*, 1(1):72–90, 1900. Available online on JSTOR.
194. G. M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. Technical report, IBM Ltd., Ottawa, Ontario, 1966.
195. R. D. Nair, H.-W. Choi, and H. M. Tufo. Computational aspects of a scalable high-order discontinuous Galerkin atmospheric dynamical core. *Computers & Fluids*, 38(2):309–319, 2009.
196. R. Niedermeier, K. Reinhardt, and P. Sanders. Towards optimal locality in mesh-indexings. *Discrete Applied Mathematics*, 117:211–237, 2002.
197. M. G. Norman and P. Moscato. The Euclidian traveling salesman problem and a space-filling curve. *Chaos, Solitons & Fractals*, 6:389–397, 1995.
198. A. Null. Space-filling curves, or how to waste time with a plotter. *Software: Practice and Experience*, 1:403–410, 1971.
199. Y. Ohno and K. Ohyama. A catalog of non-symmetric self-similar space-filling curves. *Journal of Recreational Mathematics*, 23(4):247–254, 1991.
200. Y. Ohno and K. Ohyama. A catalog of symmetric self-similar space-filling curves. *Journal of Recreational Mathematics*, 23(3):161–174, 1991.
201. M. A. Oliver and N. E. Wiseman. Operations on quadtree encoded images. *The Computer Journal*, 26(1):83–91, 1983.
202. J. A. Orenstein. Spatial query processing in an object-oriented database system. *ACM SIGMOD Record*, 15(2):326–336, 1986.
203. J. A. Orenstein and F. A. Manola. PROBE spatial data modeling in an image database and query processing application. *IEEE Transactions on Software Engineering*, 14(5):611–629, 1988.
204. J. A. Orenstein and T. H. Merrett. A class of data structures for associative searching. In *Proceedings of the 3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 181–190. ACM, 1984.
205. C.-W. Ou, M. Gunwani, and S. Ranka. Architecture-independent locality-improving transformations of computational graphs embedded in k-dimensions. In *ICS '95: Proceedings of the 9th International Conference on Supercomputing*, pages 289–298, 1995.
206. C.-W. Ou, S. Ranka, and G. Fox. Fast and parallel mapping algorithms for irregular problems. *Journal of Supercomputing*, 10:119–140, 1996.
207. R. Pajarola. Large scale terrain visualization using the restricted quadtree triangulation. In *VIS '98: Proceedings of the Conference on Visualization '98*, pages 19–26. IEEE Computer Society Press, 1998.
208. S. Papadomanolakis, A. Ailamaki, J. C. Lopez, T. Tu, D. R. O'Hallaron, and G. Heber. Efficient query processing on unstructured tetrahedral meshes. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 551–562, 2006.

209. M. Parashar, J. C. Browne, C. Edwards, and K. Klimkowski. A common data management infrastructure for parallel adaptive algorithms for PDE solutions. In *Proceedings of the 1997 ACM/IEEE Conference on Supercomputing*, pages 1–22. ACM Press, 1997.
210. M. Parashar and J.C. Browne. On partitioning dynamic adaptive grid hierarchies. In *Proceedings of the 29th Annual Hawaii International Conference on System Sciences*, pages 604–613, 1996.
211. V. Pascucci. Isosurface computation made simple: Hardware acceleration, adaptive refinement and tetrahedral stripping. In *Joint Eurographics-IEEE TVCG Symposium on Visualization (VisSym)*, pages 293–300, 2004.
212. A. Patra and J.T. Oden. Problem decomposition for adaptive *hp* finite element methods. *Computing Systems in Engineering*, 6(2):97–109, 1995.
213. A. K. Patra, A. Laszloffy, and J. Long. Data structures and load balancing for parallel adaptive hp finite-element methods. *Computers & Mathematics with Applications*, 46(1):105–123, 2003.
214. G. Peano. Sur une courbe, qui remplit toute une aire plane. *Mathematische Annalen*, 36:157–160, 1890. Available online on the Göttinger Digitalisierungszentrum.
215. A. Pérez, S. Kamata, and E. Kawagutchi. Peano scanning of arbitrary size images. In *11th IAPR International Conference on Pattern Recognition, 1992. Vol.III. Conference C: Image, Speech and Signal Analysis*, pages 565–568. IEEE, 1992.
216. E. Perlman, R. Burns, Y. Li, and C. Meneveau. Data exploration of turbulence simulations using a database cluster. In *SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, pages 1–11, 2007.
217. J. R. Pilkington and S. B. Baden. Partitioning with spacefilling curves. CSE Technical Report Number CS94–349, University of California, San Diego, 1994.
218. J. R. Pilkington and S. B. Baden. Dynamic partitioning of non-uniform structured workloads with spacefilling curves. *IEEE Transactions on Parallel and Distributed Systems*, 7(3):288–300, 1996.
219. L. K. Platzman and J. J. Bartholdi III. Spacefilling curves and the planar travelling salesman problem. *Journal of the ACM*, 36(4):719–737, 1989.
220. G. Polya. Über eine Peanosche Kurve. *Bulletin de l'Académie des Sciences de Cracovie, Série A*, pages 1–9, 1913.
221. P. Prusinkiewicz. Graphical applications of L-systems. In *Proceedings of Graphics Interface '86/Vision Interface '86*, pages 247–253, 1986.
222. P. Prusinkiewicz and A. Lindenmayer. *The Algorithmic Beauty of Plants*. Springer, 1990.
223. P. Prusinkiewicz, A. Lindenmayer, and F. D. Fracchia. Synthesis of space-filling curves on the square grid. In *Fractals in the Fundamental and Applied Sciences*, pages 341–366. North Holland, Elsevier Science Publisher B.V., 1991.
224. J. Quinqueton and M. Berthod. A locally adaptive Peano scanning algorithm. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-3(4):403–412, 1981.
225. A. Rahimian, I. Lashuk, S.K. Veerapaneni, A. Chandramowlishwaran, D. Malhotra, L. Moon, R. Sampath, A. Shringarpure, J. Vetter, R. Vuduc, D. Zorin, and G. Biros. Petascale direct numerical simulation of blood flow on 200k cores and heterogeneous architectures. In *ACM/IEEE Conference on Supercomputing, 2010*, pages 1–11, 2010.
226. F. Ramsak, V. Markl, R. Fenk, Elhardt K., and R. Bayer. Integrating the UB-tree into a database system kernel. In *Proceedings of the 26th International Conference on Very Large Databases*, pages 263–272, 2000.
227. M.-C. Rivara and Ch. Levin. A 3-d refinement algorithm suitable for adaptive and multi-grid techniques. *Communications in Applied Numerical Methods*, 8:281–290, 1992.
228. S. Roberts, S. Kalyanasundaram, M. Cardew-Hall, and W. Clarke. A key based parallel adaptive refinement technique for finite element methods. In *Computational Techniques and Applications: CTAC 97*, pages 577–584. World Scientific Press, 1998.
229. B. Roychoudhury and J.F. Muth. The solution of travelling salesman problems based on industrial data. *Journal of Complexity*, 46(3):347–353, 1995.

230. H. Sagan. Some reflections on the emergence of space-filling curves. *Journal of the Franklin Institute*, 328:419–430, 1991.
231. H. Sagan. On the geometrization of the Peano curve and the arithmetization of the Hilbert curve. *International Journal of Mathematical Education in Science and Technology*, 23(3):403–411, 1992.
232. H. Sagan. A three-dimensional Hilbert curve. *International Journal of Mathematical Education in Science and Technology*, 24(4):541–545, 1993.
233. H. Sagan. *Space-Filling Curves*. Universitext. Springer, 1994.
234. J. K. Salmon, M. S. Warren, and G. S. Winckelmans. Fast parallel tree codes for gravitational and fluid dynamical n-body problems. *International Journal of Supercomputer Applications*, 8(2):129–142, 1994.
235. H. Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys*, 16(2):187–260, 1984.
236. P. Sanders and T. Hansch. Efficient massively parallel quicksort. In *Proceedings of the 4th International Symposium on Solving Irregularly Structured Problems in Parallel*, volume 1253 of *Lecture Notes in Computer Science*, pages 13–24, 1997.
237. M. Saxena, P. M. Finnigan, C. M. Graichen, A. F. Hathaway, and V. N. Parthasarathy. Octree-based automatic mesh generation for non-manifold domains. *Engineering with Computers*, 11(1):1–14, 1995.
238. S. Schamberger and J.-M. Wierum. A locality preserving graph ordering approach for implicit partitioning: Graph-filling curves. In *Proceedings of the 17th International Conference on Parallel and Distributed Computing Systems, (PDCS'04)*, pages 51–57. ISCA, 2004.
239. S. Schamberger and J.-M. Wierum. Partitioning finite element meshes using space-filling curves. *Future Generation Computer Systems*, 21:759–766, 2005.
240. K. Schloegel, G. Karypis, and V. Kumar. *Graph partitioning for high-performance scientific simulations*, pages 491–541. Morgan Kaufmann Publishers Inc., 2003.
241. W.J. Schroeder and M.S. Shephard. A combined octree/Delaunay method for fully automatic 3-d mesh generation. *International Journal for Numerical Methods in Engineering*, 26(1):37–55, 1988.
242. E. G. Sewell. Automatic generation of triangulation for piecewise polynomial approximation. Technical Report CSD-TR83, Purdue University, 1972. PhD Thesis.
243. M.S. Shephard and M.K. Georges. Automatic three-dimensional mesh generation by the finite octree technique. *International Journal for Numerical Methods in Engineering*, 32(4):709–749, 1991.
244. W. Sierpinski. Sur une nouvelle courbe continue qui remplit toute une aire plane. *Bulletin de l'Académie des Sciences de Cracovie, Série A*, pages 462–478, 1912.
245. J. P. Singh, C. Holt, T. Totsuka, A. Gupta, and J. Hennessy. Load balancing and data locality in adaptive hierarchical n-body methods: Barnes–Hut, fast multipole, and radiosity. *Journal of Parallel and Distributed Computing*, 27:118–141, 1995.
246. R. Siromoney and K. G. Subramanian. Space-filling curves and infinite graphs. In *Graph-Grammars and Their Application to Computer Science*, volume 153 of *Lecture Notes in Computer Science*, pages 380–391, 1983.
247. B. Smith, P. Bjørstad, and W. Gropp. *Domain Decomposition: Parallel Multilevel Methods for Elliptic Partial Differential Equations*. Cambridge University Press, 1996.
248. V. Springel. The cosmological simulation code GADGET-2. *Monthly Notices of the Royal Astronomical Society*, 364:1105–1134, 2005.
249. J. Steensland, S. Chandra, and M. Parashar. An application-centric characterization of domain-based SFC partitioners for parallel SAMR. *IEEE Transactions on Parallel and Distributed Systems*, 13(12):1275–1289, 2002.
250. R. J. Stevens, A. F. Lehar, and F. H. Preston. Manipulation and presentation of multidimensional image data using the Peano scan. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-5(5):520–526, 1983.
251. Q. F. Stout. Topological matching. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, pages 24–31, 1983.

252. H. Sundar, R. S. Sampath, S. S. Adavani, C. Davatzikos, and G. Biros. Low-constant parallel algorithms for finite element simulations using linear octrees. In *SC '07: Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, pages 1–12. ACM, 2007.
253. H. Sundar, R. S. Sampath, and G. Biros. Bottom-up construction and 2:1 balance refinement of linear octrees in parallel. *SIAM Journal on Scientific Computing*, 30(5):2675–2708, 2008.
254. S. Tanimoto and T. Pavlidis. A hierarchical data structure for picture processing. *Computer Graphics and Image Processing*, 4:104–119, 1975.
255. J. Thiyagalingam, O. Beckmann, and P. H. J. Kelly. Is Morton layout competitive for large two-dimensional arrays yet? *Concurrency and Computation: Practice and Experience*, 18:1509–1539, 2006.
256. S. Tirthapura, S. Seal, and W. Aluru. A formal analysis of space filling curves for parallel domain decomposition. In *Proceedings of the 2006 International Conference on Parallel Processing (ICPP'06)*, pages 505–512. IEEE Computer Society, 2006.
257. H. Tropf and H. Herzog. Multidimensional range search in dynamically balanced trees. *Angewandte Informatik (Applied Informatics)*, 2:71–77, 1981.
258. T. Tu, D.R. O'Hallaron, and O. Ghattas. Scalable parallel octree meshing for terascale applications. In *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, page 4. IEEE Computer Society, 2005.
259. L. Velho and J. de Miranda Gomes. Digital halftoning with space filling curves. *ACM SIGGRAPH Computer Graphics*, 25(4):81–90, 1991.
260. L. Velho and J. Gomes. Variable resolution 4-k meshes: Concepts and applications. *Computer Graphics forum*, 19(4):195–212, 2000.
261. B. Von Herzen and A. H. Barr. Accurate triangulations of deformed, intersecting surfaces. *ACM SIGGRAPH Computer Graphics*, 21(4):103–110, 1987.
262. J. Wang and J. Shan. Space filling curve based point clouds index. In *Proceedings of the 8th International Conference on GeoComputation*, pages 551–562, 2005.
263. J. Warnock. A hidden surface algorithm for computer generated halftone pictures. Technical Report TR 4-15,, Computer Science Department, University of Utah, 1969.
264. M. S. Warren and J. K. Salmon. A parallel hashed oct-tree n-body algorithm. In *Conference on High Performance Networking and Computing, Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, pages 12–21. ACM, 1993.
265. T. Weinzierl. *A Framework for Parallel PDE Solvers on Multiscale Adaptive Cartesian Grids*. Dissertation, Institut für Informatik, Technische Universität München, 2009.
266. T. Weinzierl and M. Mehl. Peano – a traversal and storage scheme for octree-like adaptive Cartesian multiscale grids. *SIAM Journal on Scientific Computing*, 33(5):2732–2760, 2011.
267. R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.
268. J.-M. Wierum. Definition of a new circular space-filling curve – $\beta\Omega$-indexing. Technical Report TR-001-02, Paderborn Center for Parallel Computing, PC^2, 2002.
269. N. Wirth. *Algorithmen und Datenstrukturen*. Teubner, 1975.
270. N. Wirth. *Algorithms + Data Structures = Programs*. Prentice Hall, 1976.
271. D.S. Wise. Representing matrices as quadtrees for parallel processors. *Information Processing Letters*, 20(4):195–199, 1985.
272. D. S. Wise and S. Franco. Costs of quadtree representation of nondense matrices. *Journal of Parallel and Distributed Computing*, 9(3):282–296, 1990.
273. I. H. Witten and R. M. Neal. Using Peano curves for bilevel display of continuous tone images. *IEEE Computer Graphics and Applications*, pages 47–52, May 1982.
274. I. H. Witten and B. Wyvill. On the generation and use of space-filling curves. *Software: Practice and Experience*, 13:519–525, 1983.
275. Dawes W. N., S. A. Harvey, S. Fellows, N. Eccles, D. Jaeggi, and W. P Kellar. A practical demonstration of scalable, parallel mesh generation. In *47th AIAA Aerospace Sciences Meeting & Exhibit*, 2009. AIAA-2009-0981.
276. W. Wunderlich. Irregular curves and functional equations. *Ganita*, 5:215–230, 1954.
277. W. Wunderlich. Über Peano-Kurven. *Elemente der Mathematik*, 28(1):1–24, 1973.

278. K. Yang and M. Mills. Fractal based image coding scheme using Peano scan. In *Proceedings of ISCAS '88*, volume 1470, pages 2301–2304, 1988.
279. M.-M. Yau and S. N. Srihari. A hierarchical data structure for multidimensional digital images. *Communications of the ACM*, 26(7):504–515, 1983.
280. L. Ying, G. Biros, D. Zorin, and H. Langston. A new parallel kernel-independent fast multipole method. In *SC '03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*. IEEE Computer Society, 2003.
281. K. Yotov, T. Roeder, K. Pingali, J. Gunnels, and F. Gustavson. An experimental comparison of cache-oblivious and cache-conscious programs. In *Proceedings of the 19th annual ACM symposium on Parallel algorithms and architectures*, pages 93–104, 2007.
282. Y. Zhang and R. E. Webber. Space diffusion: an improved parallel halftoning technique using space-filling curves. In *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*, pages 305–312. ACM New York, 1993.
283. S. Zhou and C. B. Jones. HCPO: an efficient insertion order for incremental delaunay triangulation. *Information Processing Letters*, 93:37–42, 2005.
284. U. Ziegler. The NIRVANA code: Parallel computational MHD with adaptive mesh refinement. *Computer Physics Communications*, 179(4):227–244, 2008.
285. G. Zumbusch. On the quality of space-filling curve induced partitions. *Zeitschrift für Angewandte Mathematik und Mechanik*, 81, Suppl. 1:25–28, 2001.
286. G. Zumbusch. Load balancing for adaptively refined grids. *Proceedings in Applied Mathematics and Mechanics*, 1:534–537, 2002.
287. G. Zumbusch. *Parallel Multilevel Methods: Adaptive Mesh Refinement and Loadbalancing*. Vieweg+Teubner, 2003.

Index

Access locality function, 206
    Peano matrix multiplication, 208
Adaptive grids, 145
Adaptive refinement, 9
Algorithm
    vertex labelling, 89
Applications (of SFC), 235–238
Approximating polygon
    definition, 67
    entry and exit points, 20
    generator, 69
    Hilbert curve, 19, 68
        3D, 110, 111
    length, 69–72
    Peano curve, 68
    quasi-Sierpinski curve, 183
    Sierpinski curve, 79
Arithmetic representation, 94
    $\beta\Omega$-curve, 103–104
    Hilbert curve, 47–49
        3D, 113
    Peano curve, 57–59
    Sierpinski curve, 80–81

Bänsch-Kossaczky scheme, 186, 188, 192
Barnes-Hut algorithm, 140
Basic patterns, 31
    Hilbert curve, 31
        3D, 110
    Peano curve
        3D, 116
$\beta\Omega$-curve, 101–104
    arithmetisation, 103–104
    entry and exit points, 101
    grammar, 101–102
    iterations, 101
    locality properties, 107
Bézier curves and surfaces, 1
Bially's algorithm, 64
Bijective, 11
Bisection
    of tetrahedra, 124, 184–186, 191–192
    of triangles, 83
Bisection refinement, 141
Bitstream
    for modified depth-first traversal, 154
Bitstream encoding, 140, 225
    for parallel traversals, 153–155
Blocking, 13, 202
Block layout (for matrices), 213
Boundary-extended spacetrees, 232

Cache
    associative, 197
    direct-mapped, 198
    first-, second-, third-level, 196
    ideal cache, 209
    L1, L2, L3, 196
    n-associative, 197
    prefetching, 198
    replacement strategy, 198
Cache-aware, 213
    algorithms, 198
Cache lines, 197
Cache memory, 195
    hierarchy, 195, 196
    and locality, 198–199
Cache-oblivious, 213
    algorithms, 199, 233
Canonical tetrahedron, 186, 192

M. Bader, *Space-Filling Curves*, Texts in Computational Science and Engineering 9,
DOI 10.1007/978-3-642-31046-1, © Springer-Verlag Berlin Heidelberg 2013

Cantor, G., 16, 17
Cantor Set, 97, 170
Cantor's mapping, 16–17
  bijectivity, 16
Cartesian grid, 7, 8, 144
Cartesian mesh, 2
Characteristic function, 2
Closed curve, 24, 101
Clustering, 13
  of data, 12
Colour edge, 224
Colour stacks, 219
  rules, 222–225
Column-major, 12, 200
Combinatorial problems, 236
Communication, 144, 145
Communication pattern
  master-slave, 152
Compactness, 170
  partitions, 145
Computational fluid dynamics, 141, 163
Computer aided design, 1
Computer graphics, 1, 141, 233, 237
Conforming grid, 226
Conforming refinement
  spacetree grid, 226–227
Conforming triangular grid, 142
Congruency classes, 188, 192
Connected graph, 172
Connected partitions, 174
Connected SFC, 24, 107, 160, 168, 170, 174
  definition, 94
Contiguous, 11
Continuity, 12
  of space-filling curves, 24
Curve
  definition, 17
  parameter representation, 17

Data base applications, 179
Data bases, 237
Data compression, 237
Data structures
  arrays, 12
  matrices, 12
  multidimensional data, 10
Daxpy operation, 197
Delauney triangulation, 238
Depth-first traversal, 3, 133, 134, 137
  algorithm, 5
Diffusion approach
  for load balancing, 149
Dimension of fractal curves, 71
Discontinuous Galerkin methods, 218
Discrete locality measure, 171
Distributed memory, 153, 155
Dithering, 238
Domain decomposition, 158, 160
Dual graph, 172, 193
Dynamically adaptive, 145
Dynamically adaptive grids, 228

Edge-connected, 94, 182
Edge cut, 172
Element-based discretisation, 217–218
Entry and exit points
  of partitions, 162
  of SFC, 75
Entry edge, 223
Error estimation, 227
Error estimator/indicator, 145
Euclidian distance, 171
Exit edge, 223

Face-connected, 94, 173, 188, 229
Finite Element methods, 163, 181, 188, 218
Finite state machine, 62, 64
Finite Volume methods, 218
First-access order, 220
Fractal curve, 70, 72, 107
Fractal dimension, 72

Generalised Sierpinski curve, 82–88
  algorithm, 85
  circle-filling, 88
  congruency classes of triangles, 85
  continuity, 85, 87
  definition, 84
  grammar, 86
  locality, 86
  triangle-filling, 85
  on triangles with curved edges, 87
Generator, 69
  Gosper island, 107
  Koch curve, 70
Generic Space-filling Heuristic, 235
Geographical information system, 237
Geometry modelling, 1
  surface-oriented, 1
  volume-oriented, 2
Geoscience applications, 163
Ghost cells, 158, 159
  Hilbert order, 159, 160
  refinement-tree grid, 159–160

Global refinement edge
    of a tetrahedron, 186
Gosper, W., 107
Gosper curve, 104–107
    grammar, 106
    variants, 108
Gosper flowsnake. *See* Gosper curve
Gosper island, 74, 106, 107
Grammar, 94
    $\beta\Omega$-curve, 101–102
    context-free, 136
    derivation rule, 32
    generated strings, 33
    Gosper curve, 106
    Hilbert curve, 31–36
        3D, 114–116
    H-index, 99
    non-terminal symbols, 31
    Peano curve, 37–38
        3D, 119
    production rules, 32
    quasi-Sierpinski curve, 182
    shape grammars, 43
    Sierpinski curve, 79–80, 86
    table-based implementation, 44
    terminal productions, 34
    terminal symbols, 31
    turtle graphics, 39
Granularity of partitions, 152
Graph-filling curves, 193
Graph partitioning, 172
    algorithms, 173
    connected, 172
    edge cut, 172
    index-based, 162
    locality measure, 172–177

Halo cells, 159
Hamiltonian Path, 193
Hanging nodes, 157, 181, 226
Hash functions, 162
Hash table, 217
Hausdorff dimension, 72
Heat equation, 7–8, 143, 215
    residual computation, 216
    stationary problem, 7
    system of linear equations, 8, 143, 216
Hilbert, D., 17
Hilbert curve
    approximating polygon, 19, 68
        length, 69
    arithmetisation, 47–49
    basic patterns, 31, 33
    construction, 18
    continuity, 23–24, 67
    definition, 21
    fractal dimension, 71
    grammar, 31–33, 133
        adaptive, 135–136
        context-free, 136
        with terminal productions, 34–36
        turtle graphics, 39–42
    higher-dimensional, 126
    iterations, 18
    as limit curve, 19
    mapping (*see* Hilbert mapping)
    surjectivity, 22
    3D (*see* 3D Hilbert curve)
    traversal algorithm (*see* Hilbert traversal)
    turtle grammar, 222
Hilbert index, 56
    algorithm, 57, 64
    operators, 56
    3D, 113
    uniqueness, 56
Hilbert mapping, 21, 22
    algorithm, 51–52, 64
        vertex labelling, 89
    finite quaternaries, 50–51
    finite state machines, 62
    infinite quaternaries, 50, 53–55
    inverse (*see* Hilbert index)
    non-recursive implementation, 63
    operators, 49
        3D, 113
    recursion unrolling, 60
    3D, 113
    uniqueness, 52–55
Hilbert-Moore curve, 24
Hilbert order, 6
    adaptive spacetree, 135
    matrix-vector multiplication, 200
    for optimisation, 141
    quadtree, 6, 137
Hilbert traversal, 34–36
    adaptive algorithm, 136
        with bitstream encoding, 137
    adaptive spacetree, 135
    call tree, 134
    recursion unrolling, 36
    turtle-based, 42
H-index, 99–101
    grammar, 99
    iterations, 99
    locality properties, 107

Hölder continuity, 167–170, 178
  parallelisation, 168–170
  partition shape, 169
  quasi-Sierpinski curve, 184
  surface-to-volume ratio, 173
  3D Hilbert curve, 168
  3D quasi-Sierpinski curve, 190
  3D Sierpinski curve, 184
Hölder continuous
  definition, 167
H-order. *See also* H-index
  vs. Sierpinski curve, 100

Ideal cache, 209
Image compression, 140
Image data base, 179
Image processing, 238
Index
  based on SFC, 179
  computation, 28
    for partitioning, 146
Input stream, 219
Inverse mapping, 146
Inversion property, 220, 229
  Morton order, 231
Iterations
  $\beta\Omega$-curve, 101
  definition, 18
  Hilbert curve, 18
  H-index, 99
  Morton order, 95
  Peano curve, 25
  Sierpinski curve, 78
  Z-curve, 96

Join traversal, 156
  algorithm, 158

k^d-spacetree. *See also* Spacetree
  definition, 129
Kd-trees, 237
K-Median problem, 236
Knopp, K., 63, 77
Koch curve, 63, 70, 72, 74
  construction, 70
  length, 70–71
Koch snowflake, 73

Last-access order, 220
Last-in-first-out, 219
Least frequently used, 198
Least recently used, 198, 209
Lebesgue curve, 97–98, 138
  continuity, 98
  definition, 97
  vs. Morton order and Z-Curve, 97
Left-right splitting (via SFC), 161
Length of coast lines, 72, 73
Load balancing, 9, 145, 149
  diffusion approach, 149
Load distribution, 144
  exchange subgrid, 155
  subtree-based, 150–153
Locality measures, 177
  discrete, 171
  graph partitions, 172–177
  for index/inverse mapping, 179
  for iterations of SFC, 171
  partitions, 178
Locality of data, 12
Locality preserving, 94, 146
Locality properties, 185
  $\beta\Omega$-curve, 107
  H-index, 107
Local refinement edge
  of a tetrahedron, 186
Longest-edge bisection, 192
Loop unrolling, 202
L-systems, 33, 43
LU-decomposition, 212

Mandelbrot, B., 73
Manhattan distance, 171
Mapping
  computation, 28
  Hilbert curve (*see* Hilbert mapping)
  Peano curve (*see* Peano mapping)
  Sierpinski curve (*see* Sierpinski mapping)
Master-slave structure, 152
Matrix multiplication
  algorithm, 202
  blocking and tiling, 212
  Peano curves (*see* Peano matrix multiplication)
  as 3D traversal, 202
Matrix operations, 163
Matrix storage, 213
  conversion of formats, 213
Matrix-vector multiplication, 199–201
  algorithm, 199, 200
    Hilbert traversal, 201
  cache efficiency, 199–201
  Hilbert order, 200

Maximum distance, 171
Memory-bound performance, 196–197
Memory gap, 195
Modified depth-first traversal, 153–155
  algorithm, 156
Molecular dynamics, 163, 233
Moore, E. H., 24, 29
Morton order, 94–96, 107, 138, 140, 229–232
  construction, 95
  inversion property, 231
  iterations, 95
  non-continuous, 95
  for optimisation, 141
  projection property, 229
  quadtree, 139
Multicore CPUs, 196
Multidimensional arrays, 10
Multidimensional data, 9
  algorithms and operations, 10
Multigrid method, 9

N-body problem, 162
Nearest-neighbour problem, 237
Neighbour relations, 12, 13, 146
Nested intervals, 21, 48
Netto, E., 17
Newest vertex bisection, 89, 141
Node-connected, 94, 173, 182, 189
Non-terminal symbols. *See* Grammar
Non-uniform memory access, 196
N-order, 96
Norm cell scheme, 2, 131
  number of cells, 2
n-tuple, 10
NUMA. *See* Non-uniform memory access
Numerical linear algebra, 13
NURBS, 1

Octree, 3, 4, 107, 129
  computer graphics, 140
  grid generation, 140
  number of grid cells, 132
  2^3-spacetree, 130
Old/new classification of edges, 224
Output stream, 220

Padding, 210
Palindrome property, 228, 232
Parameterised by volume, 169, 191
  Lebesgue/Peano/Sierpinski curve, 170
  3D Hilbert curve, 169
Particle-base simulation, 163
Partition boundaries
  left and right part, 161
  subtree-based partitioning, 152
Partitioning, 9, 10
  criteria for efficiency, 144–146
  index-based, 146–147
  parallel sorting, 147
  refinement-tree, 149–150
    parallel algorithm, 150
  sequentialised refinement trees, 153–156
  software, 163
  space-filling curves, 146–156, 162, 173
  subtree-based, 150–153, 162
  traversal-based, 148–149
    unstructured grids, 148
Partitions, 144
  compact, 145, 170
  connected, 173–177, 179
  data exchange, 157–162
  disconnected, 174
  length of boundary, 145
  locality measures, 178
  Morton order, 175–177
  number of unknowns, 144
  spacetree, 175
  surface-to-volume ratio, 173, 178
Peano, G., 17, 25
Peano curve
  approximating polygon, 68
  arithmetisation, 57–59
  construction, 25
  construction by Peano, 122
  continuity, 27
  dimension recursive, 116–117, 122
  fractal dimension, 71
  grammar, 37–38
    dimension recursive, 116–117
  iterations, 25
    grids of arbitrary size, 120–122
  mapping (*see* Peano mapping)
  Meurthe order, 29
  notation, 29
  projection property, 232
  5×5 or 7×7 refinement, 119
  surjectivity, 26
  switch-back type, 120
  3D (*see* 3D Peano curve)
  traversal algorithm (*see* Peano traversal)
  variants, 29
    Meander type (*see* Peano-Meander curve)
    switch-back type, 26
Peano index, 146

Peano iterations
    grids of arbitrary size, 210
Peano mapping
    algorithm, 59
    dimension recursive, 122
    operators, 57–59
    Peano's original formulation, 122
Peano matrix multiplication, 201–210
    block-recursive scheme, 204–206
    cache efficiency, 206–210
    cache misses, 208–210
    contiguous access, 208
    increment/decrement access, 205, 206
    matrices of arbitrary size, 210
    memory access pattern, 205–206
        locality properties, 206
    parallelisation, 210, 212
    recursive blocking, 208
    recursive implementation, 207
    3 × 3-scheme, 203
Peano-Meander curve, 26
Peano order
    grids of arbitrary size, 120–122
    for matrix elements, 204
    spacetree, 139
Peano scan, 237
Peano traversal, 38, 232
    grids of arbitrary size, 121
Performance
    daxpy operation, 197
    matrix multiplication, 197
    memory-bound, 196–197
Plotter, 235
Production rules. *See* Grammar
Projection property, 232
    Morton order, 229

Quadtree, 3–7, 107, 129, 137, 140, 213
    bitstream encoding, 138, 140
    boundary cells, 131
    construction, 3, 4
    depth-first traversal, 5
    ghost cells, 159
    Hilbert order, 6, 138, 140
    Morton order, 139
    number of grid cells, 131–132, 140
    restricted quadtree, 141
    sequentialisation by Hilbert curve, 133
    sequential order, 3–7
    2^2-spacetree, 130
    traversal, 3
    Z-order, 5
Quasi-Sierpinski curve, 182–184
    algorithm, 183
    approximating polygon, 183
    grammar, 182
    Hölder continuity, 184
    mapping, 183
    3D, 189–191
        Hölder continuity, 190–191
Quasi-Sierpinski order
    on triangular meshes, 184
Quaternary representation, 48
Queue data structures, 230
Queue property
    (violation by) Morton order, 231

Range queries, 237
Recursion unrolling, 60–62
Recursive blocking (for matrix storage), 213
Recursive SFC, 24, 107, 168, 170
    definition, 93
Red-black refinement, 186
Red-green refinement, 181
Refinement bit, 134
Refinement cascade, 226
Refinement tree, 138, 141, 162, 175, 193, 225
    parallel grid partitions, 155–156
Refinement-tree partitioning, 149–150
Residual computation
    element-based, 218
Residuals, 216
Row-major, 12, 199
R-trees, 237
Runtime complexity, 196

Sagan, H., 28
Self-avoiding walks, 193
Self-similar, 94
Self-similarity, 48
Separator, 160
Sequentialisation, 9–13
    column-major, 12
    row-major, 12
Sequentialisation by SFC, 27
Sequential order, 3, 146
    family, 11
    Hilbert curve, 6
    locality, 6–7
    requirements, 11–12
Shared memory, 153
Sierpinski, W., 77
Sierpinski curve, 100
    approximating polygon, 79

  - arithmetisation, 80–81
  - construction, 77
  - definition, 77
  - generalised curve (*see* Generalised Sierpinski curve)
  - grammar, 79–80, 100
  - vs. H-order, 100
  - iterations, 78
  - mapping (*see* Sierpinski mapping)
  - node-connected (*see* Quasi-Sierpinski curve)
  - parallelisation, 162
  - 3D (*see* 3D Sierpinski curve)
  - turtle grammar, 223
- Sierpinski-Knopp curve, 77
- Sierpinski mapping, 63, 81–82
  - algorithm, 82
  - non-recursive implementation, 82
  - operators, 81
- Sierpinski order
  - adaptive triangular grid, 140
  - red-green refinement, 184
  - triangle strips, 141
- Signal processing, 237
- Simple SFC, 107, 109
  - definition, 94
- Smoothed Particle Hydrodynamics, 163
- Space-filler, 94, 107
- Space-filling curve
  - connected (*see* Connected SFC)
  - definition, 17
  - recursive (*see* Recursive SFC)
- Spacetree, 175, 211
  - adaptive, 130
  - adaptive traversal, 134–135
  - definition, 129
  - k^d-spacetree, 129
  - numerical simulation on spacetree grids, 215–232
  - Peano order, 139
  - regularly refined, 130
  - sequentialisation, 132
    - Hilbert order, 135
- Spacetree grid
  - access to neighbour cells, 216–217
  - conforming refinement, 226–227
  - ghost cells, 159–160
- Spacetree traversal
  - element-based, 217–219
  - first-access order, 220
  - last-access order, 220
  - Morton order, 229–232
  - multiple access to unknowns, 218
  - Peano order, 232
  - stack-and-stream scheme, 220
- Sparse matrix, 211
  - Hilbert order, 213
  - Peano order, 211
  - quadtree storage scheme, 213
  - spacetree storage scheme, 211–212
- Spatial locality, 198
- Split traversal, 155
- Stack-based traversal
  - adaptive grid, 226
  - algorithm, 225–226
  - edge-located unknowns, 227, 254
  - memory efficiency, 227–228
  - old/new classification, 223–224
  - Peano curves, 229
  - stack rules, 222–225
- Stack property, 219, 228
  - Hilbert curve, 219
  - partition boundaries, 161
  - Peano curve, 221
  - Sierpinski curve, 221
  - (violation by) Hilbert order, 229
- Surface-oriented geometry modelling, 1
- Surjective, 17

- Tagged edge, 83, 124
- Temporal locality, 198
- Terminal symbols. *See* Grammar
- Tetrahedral grids, 181, 184–191
  - angles of tetrahedra, 188
  - bisection refinement, 184–188, 191–192
    - shapes of tetrahedra, 186
  - longest-edge bisection, 192
  - red-black refinement, 187, 192
- Tetrahedral meshes. *See* Tetrahedral grids
- Tetrahedral strips, 141, 233
- Tetrahedron with tagged edge, 124
- 3D Hilbert curve, 109–116
  - approximating polygon, 110, 111
  - arithmetisation, 113
  - basic patterns, 110
    - rotation, 111, 112
  - face-connected, 109
  - grammar, 114–116
    - number of terminals, 114
  - Hölder continuity, 168
  - mapping, 113
  - operators, 113
  - parameterised by volume, 169
  - variants, 109–112, 126
    - number of different curves, 112

3D Hilbert index, 113
3D Hilbert traversal
  palindrome/stack property, 229, 230
3D Peano curve, 116–119
  basic patterns, 116
  dimension recursive, 116, 117
  grammar, 119
  projection property, 202, 203
  switch-back type, 116
3D Sierpinski curve, 123–125
  algorithm, 125
  definition, 124
  face-connected, 125
  Hölder continuity, 184
  tetrahedral strips, 141
3D Sierpinski order, 192
Tiling, 13, 202
Topological monsters, 235
Translation lookaside buffer, 213
Travelling salesman problem, 28, 236
Traversal, 3, 10
  on adaptive spacetrees, 134–135
  algorithm, 34
    Hilbert curve (*see* Hilbert traversal)
    Peano curve (*see* Peano traversal)
  computational costs, 36
  depth-first (*see* Depth-first traversal)
  of a matrix, 200
  modified depth-first, 153
  in SFC order, 28
  turtle-based vs. plotter-based, 42
Tree algorithms, 163
Triangle strips, 141, 233
  swap command, 142
Triangle with tagged edge, 83
Triangular grid, 8, 144, 181–184
  adaptive, 141
  red-green refinement, 181–184
  Sierpinski order, 140
Triangular meshes, 141
Tuple, 10
Turtle grammar
  Hilbert curve, 222
  Sierpinski curve, 223
Turtle graphics, 33
  $\beta\Omega$-curve, 102
  Gosper curve, 106
  grammar, 39

UB-trees, 237
Uniformly continuous, 23, 94
Uniqueness
  of inverse mapping, 56
  of SFC mappings, 52–55

Vertex labelling, 84, 89, 125, 183, 190
Volume-oriented geometry modelling, 2

Wire-frame model, 1, 2
Working set, 201
Work pool approach, 152
Work stealing, 153
Wunderlich, W., 29

Z-curve, 96, 140
  iterations, 96
  mapping, 96
Z-order. *See* Z-curve