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,

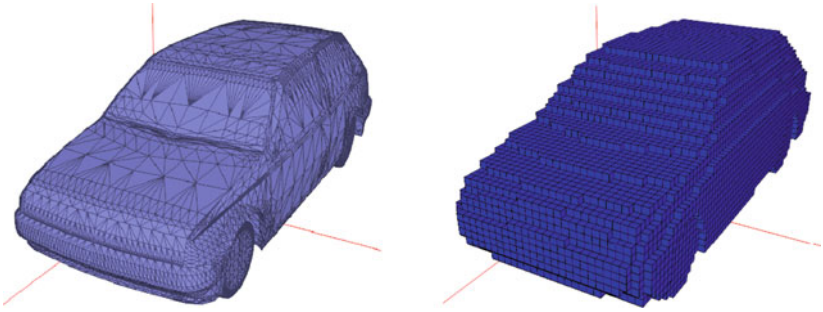


Fig. 1.1 3D geometry models of a car: surface-oriented modelling with a wire-frame model (*left image*) vs. a 2D norm-cell scheme (*right image*) (Images reproduced (with permission) from Mundani and Daubner [72])

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

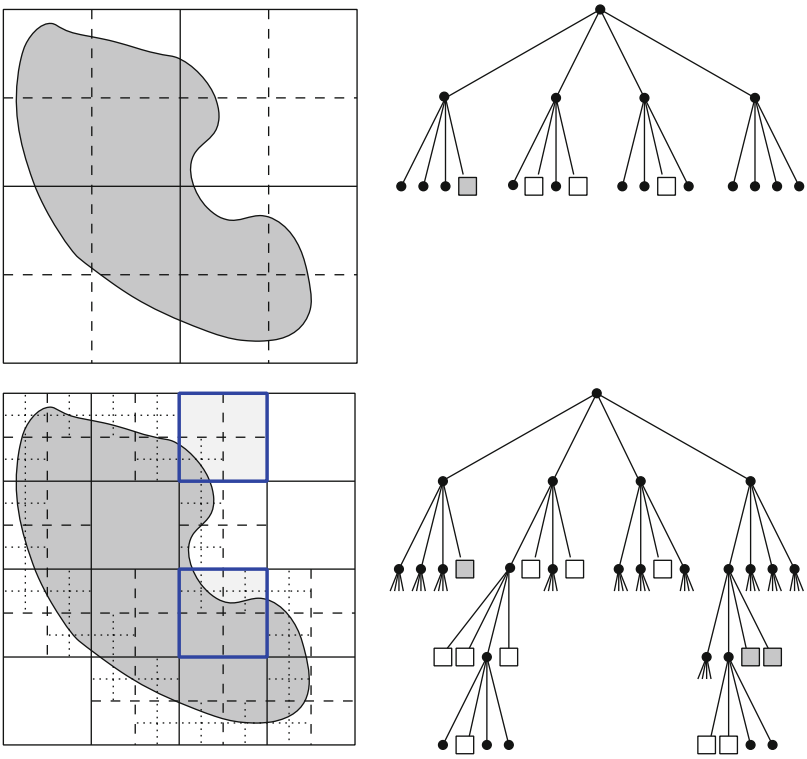


Fig. 1.2 Generating the quadtree representation of a 2D domain. The quadtree is fully extended only for the highlighted cells

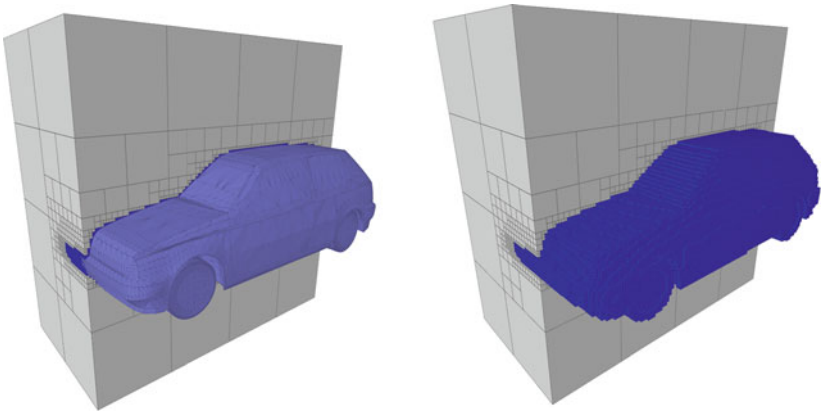


Fig. 1.3 Generating the octree representation of a car body from a respective surface model (Images with permission from Mundani and Daubner [72])

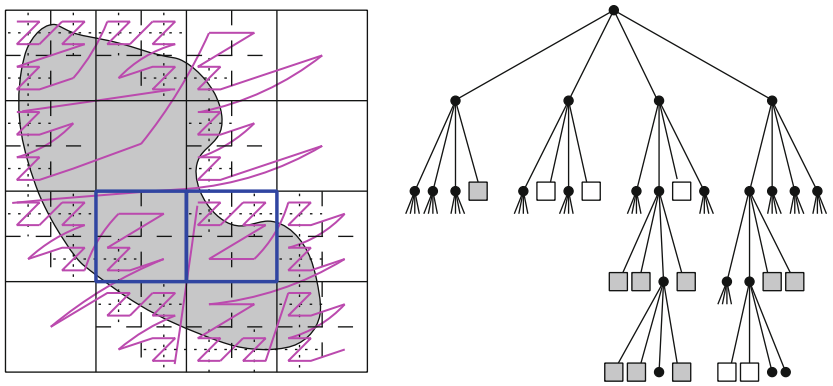


Fig. 1.4 Quadtree representation of a given object and a simple sequential order on the quadtree cells that results from a depth-first traversal of the corresponding tree structure

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.

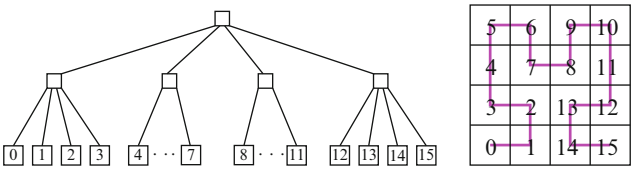


Fig. 1.5 Quadtree representation of a regular 4 × 4 grid, and a sequential order that avoids jumps

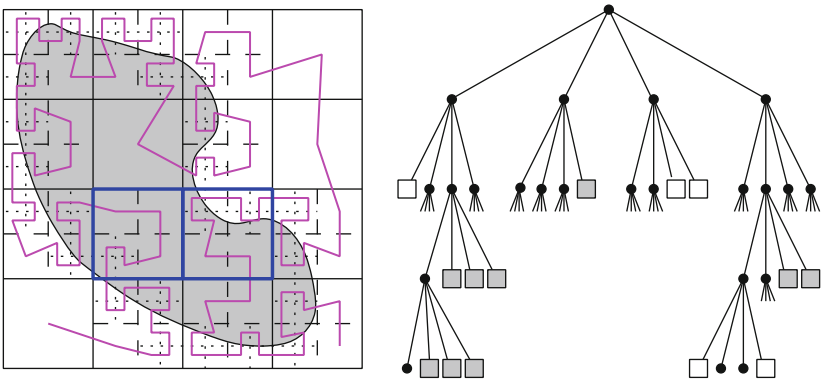


Fig. 1.6 Quadtree representation of a given object and a sequential order on the quadtree cells that avoids jumps – the order corresponds to a depth-first traversal of the corresponding tree

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

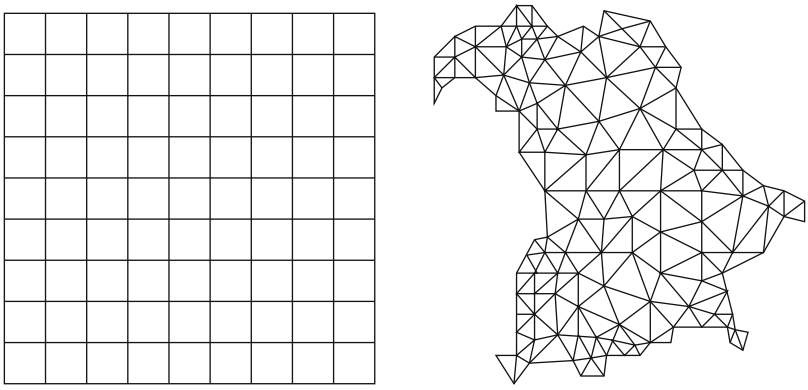
Let us move to a further example, which is motivated by the field of numerical simulation. As a simple example for such a problem, we examine the computation of the temperature distribution on a metal plate. The geometrical form of the metal plate shall be given, which also defines the computational 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.

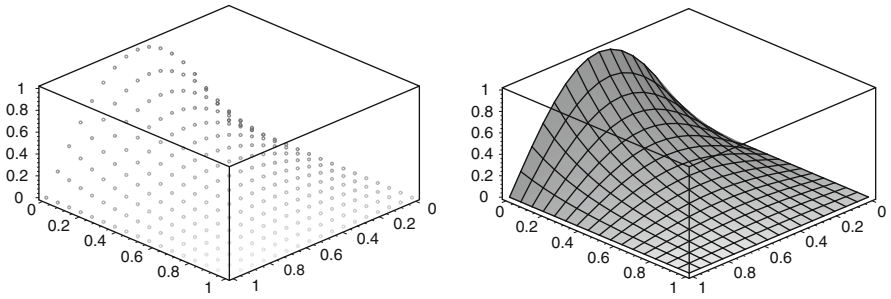


Fig. 1.8 Solution of our model problem on a 16 × 16 Cartesian grid. The *left plot* shows the temperatures at the grid points, whereas the *right plot* gives a bilinear interpolation of this pointwise solution

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, ..., n do
    for j = 1, ..., 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, . . . , n* **do**
  for *j = 1, . . . , m* **do**
    . . .

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

for all *n-tupels (i,j,. . .)* **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).

http://www.springer.com/978-3-642-31045-4

Space-Filling Curves
An Introduction with Applications in Scientific Computing
Bader, M.
2013, XIII, 285 p., Hardcover
ISBN: 978-3-642-31045-4