4 Data Modelling for Visualization

In this chapter, we discuss issues that arise when generating data that is suitable for scientific visualization. There are two scenarios that represent the majority of the cases we encounter in this book. The first is measurement: data is being acquired from a sensor that is examining properties of the physical world. The second is simulation: after a mathematical model of an entity is defined, a computer simulation generates data that respects this model (typically integrating differential equations or approximations thereof).

We start presenting the types of data that are popular in scientific visualization. We then discuss the fundamental tension between the physical world and any representation in a digital computer that arises from finite storage and processing. This discussion brings us to the subjects of sampling and interpolation. Then, we discuss the assumptions of sampling the physical world uniformly, and describe data types that avoid that and the consequences of these choices. These are sometimes called *irregular grids* or *unstructured grids* (as opposed to *regular* or *structured* grids).

4.1 Data Types

We are interested in visualizing many different aspects of the physical world, and these are represented by different types of data. Sometimes we can assign a single number to every point in space (we call these "scalar fields", §4.1.3); often, we need direction and magnitude information ("vector fields", §4.1.4); somewhat more rarely, we can associate with every point in space a linear transformation (§4.1.5. Most of the data types we will describe here can adequately be described as *mappings* between two sets: the *domain* and the range. The type of range of the mapping encodes what type of data will be available (for example, scalars, vectors and tensors). The domain, on the other hand, defines the shape of the space we're examining. If we measure temperature data throughout a city, we will typically use a rectangular domain, giving rise to a 2D scalar field. If we want to examine global temperatures, however, our domain will be the surface of a sphere, which is better characterized by latitude and longitude than a rectangular

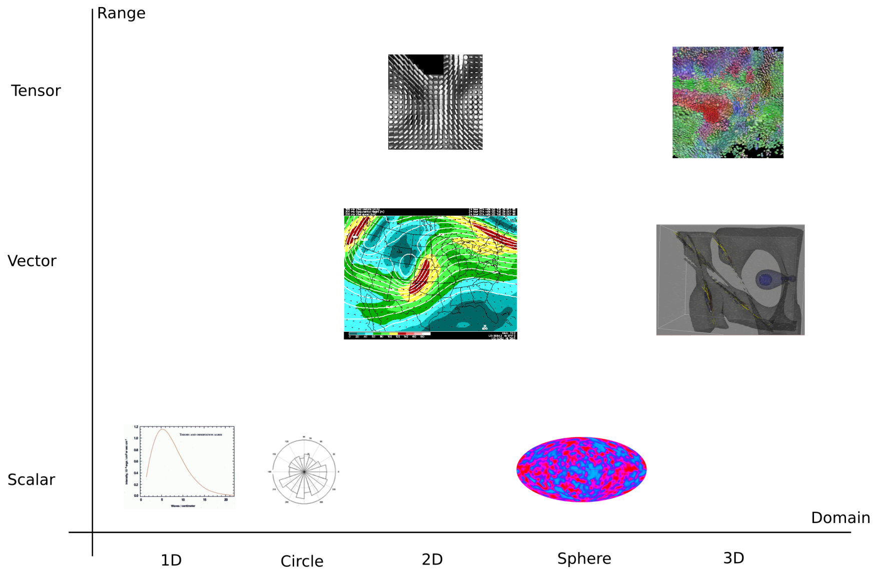

Figure 4.1. An overview of datatypes that will are used in scientific visualization, organized by domain and range distinctions.

coordinate system. Figure 4.1 illustrates the main data types.

4.1.1 The real number field vs the floating point "field"

Throughout this text, we will largely ignore one very important aspect of computational science: that a single arbitrary real number cannot be represented accurately in a computer. Instead, computers use a number system that is an approximation of the real number field (the floating point system), roughly equivalent to the standard scientific notation for numbers, with a fixed number of digits for the mantissa. There is an entire scientific area between applied mathematics and computer science called *numerical analysis* that is concerned exactly with the many and very important consequences of this assumption (for example, addition in this number system is not even associative).

In this book, we will assume that each storage unit of memory is capable of storing a single real number. In the cases where numerical issues become important for the quality or performance of results in scientific visualization (for example, the volume rendering integral can be quite sensitive to precision issues), we will make it clear.

4.1.2 Functions

We start describing "regular" functions $\mathbf{R} \rightarrow \mathbf{R}$. They might seem trivial and uninteresting, but some issues we encounter on more complicated data types can be traced back to fundamental issues in one-dimensional data, so we describe them first.

Given a function f, we are obviously interested in its function values $f(x)$ at a set of points $P \subset \mathbf{R}$. However, we also want more information, in the form of function derivatives, integrals and others. We will assume most of the time that functions are sufficiently differentiable:

$$\forall f, \exists f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

.

Function values are scalars, and its derivatives and integrals are again regular $\mathbf{R} \rightarrow \mathbf{R}$ functions, so the behavior is fairly simple. Other data types, however, do not share this simplicity, as we will see.

4.1.3 Scalar Fields

The simplest scalar fields are defined by a mapping from $\mathbf{R}^n \rightarrow \mathbf{R}$. For every point $p \in \mathbf{R}^n$, a scalar field f gives a real $f(p)$. Unlike functions, the behavior of vector scalar under differentiation is quite different. There are now n dimensions in the domain, so the limit definition of derivatives needs to be extended: there will have n different limits, one for each dimension. *The derivative of a scalar field, then, is not a scalar field.* Let us look at a simple scalar field: $f(x,y) = x^2 + y^2$, and let's examine its behavior around $x = 1, y = 1$. We can take derivatives in x and y: $\partial f / \partial y(1) = 1$, $\partial f / \partial y(1) = 1$. We then say $\nabla f(1,1) = (2,2)$. We need, however, a good geometrical interpretation of this value. It turns out that the best way to look at that pair of values as a *direction*: it is the direction in which f *changes the fastest* around $x = 1, y = 1$. The derivative of f is typically called the *gradient* of f, and it is the simpelst example of a *vector field*.

4.1.4 Vector Fields

The only difference between scalar and vector fields is the range of the mapping. Vector fields are mappings $R^n \rightarrow \mathbf{R}^n$: the range has extra dimensions. Vector fields are used in situations where each point needs to store a quantity with magnitude and direction. Each of these vectors is best imagined as an arrow. Vector fields, have much richer structure than scalar fields, specially in higher dimensions. As we will see in Chapter 8, there are different quantities we can calculate from a vector field such as the *divergence* and the *curl* that give great quantitative and qualitative insight into the nature of the vector field.

As with scalar fields, the derivative of a vector field is not a vector field either. Each partial derivative of the domain yields a vector, so the derivative at a point

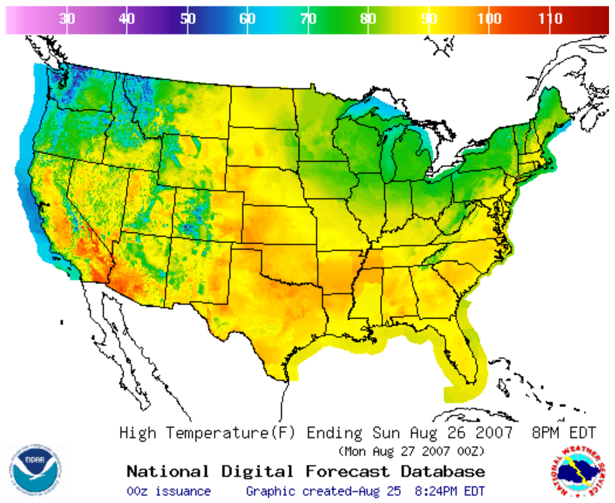
30
40
50
60
70
80
90
100
110
High Temperature(F) Ending Sun Aug 26 2007 8PM EDT
(Mon Aug 27 2007 00Z)
National Digital Forecast Database
00z issuance Graphic created-Aug 25 8:24PM EDT

Figure 4.2. A temperature map is an example of a two-dimensional scalar field.

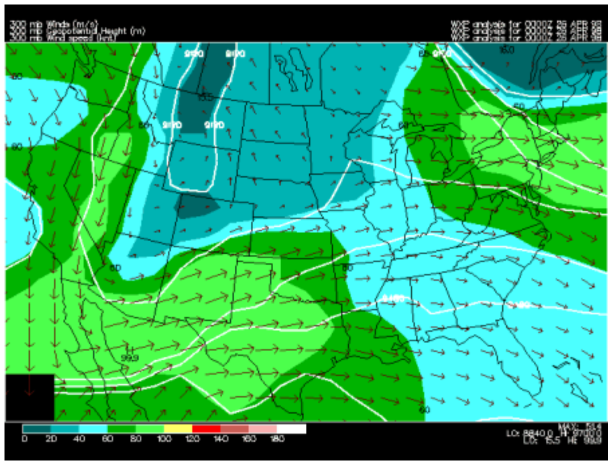

Figure 4.3. A wind map is an example of a two-dimensional vector field.

gives us an entity with n^2 values. As you have probably guessed, this turns out to be a tensor.

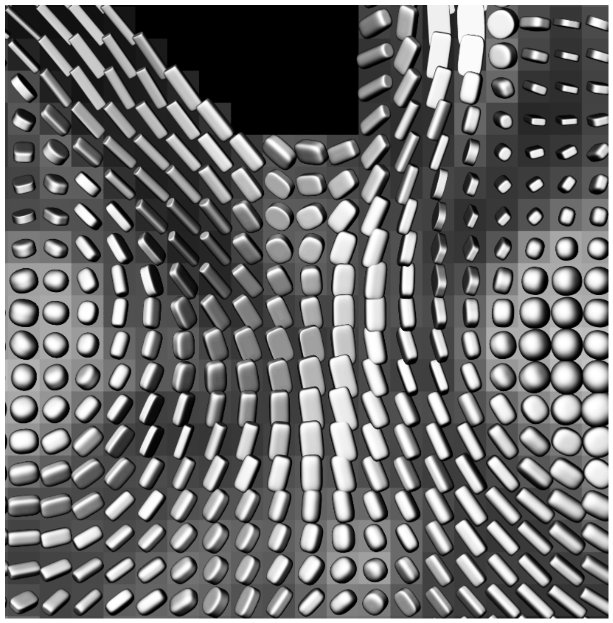

Figure 4.4. A slice through a DTI dataset that shows how water diffuses differently throughout the brain. More directional glyphs represent stronger preference towards that particular direction.

4.1.5 Tensor Fields

The most complicated range type we will see in this book are tensors. More specifically, we will be dealing with *rank-2* tensors, which are essentially matrices. In scientific visualization, tensors became a popular data modality with the advent of diffusion tensor magnetic resonance imaging: DT-MRI, or DTI for short. Just as vectors encode signficantly more information than scalars, tensors also encode more information. As we will see, in scientific visualization tensors are particularly useful to encode *anisotropy*: behavior that is stronger in some directions. In DTI image, each tensor stores the difference in water diffusivity: water tends to travel faster through neurological pathways than across them. Tensors have also been recently used in visualizing liquid crystal alignment. Both of these are a particularly simple type of tensor: they are symmetric. As we will see in chapter 8, this makes visualization and processing significantly easier.

4.1.6 More complicated domains: circles, spheres, *etc.*

All of the domains which we will see in this book are either Cartesian (R^n for some n), or the result of a simple coordinate transformation. For example, functions on circles are easiest seen as functions on an angle parameterization of the circle. This way, instead of thinking as functions on circles as a function with a limited domain $D \subset \mathbf{R}^2$, we define a new parameter t, define the function on t, and remember that to go back to our circle in two-dimensional space, we need to apply the transformations $x(t) = \cos t, y(t) = \sin t$.

These coordinate changes need to be watched out for. gradients of functions defined on circles are vectors whose coordinates are parameters in the domain. Imagine a temperature function defined on the earth. The direction of biggest change can be seen of as a 2D-vector in the latitude-longitude basis, but it can also be seen as a 3D-vector in a cartesian domain.

4.2 Continuous vs. Discrete: Sampling and Reconstruction

The real world is continuous, and so are the things we want to visualize and simulate in a computer. A computer, however, can only deal with a finite amount of data. It is very important, then, to be able to switch back and forth between these two world-views. Most of the data modelling problems and solutions we will see address this directly. We will discuss the mathematics of one-dimensional sampling and reconstruction, since most of the n-dimensional techniques are direct extensions of 1-D methods, and these are much simpler to illustrate.

4.2.1 Sampling

The easiest way to store a finite representation of a continous function is to *sample* the function values in regular intervals, and store these. To store and manipulation a function $f(x)$ in an interval $[a,b] \subset \mathbf{R}$, we define a certain sampling density δ, and store in a computer an indexed array of values $f_i = f(a + \delta(b-a)i)$, $0 \leq i \leq \delta^{-}1$. From then on, we forget about $f(x)$, and manipulate the values f_i exclusively, as shown in Figure 4.7. We will clearly have lost information about the behavior of the function outside the sampled values.

It is important to realize that most real-life sensors *do not* work in such a simple way. For example, Computed Tomography scanners use the Fourier Slice Theorem [Bracewell 99] to reconstruct a density value on each point in space while examining only the density integral along rays through the sensed object. However, this is a reasonable model to understand and analyze, so we will use it.

4.2.2 Reconstruction: Interpolation and Approximation

Now that we have a finite representation of a function, we need to go back to having a continous function. This is important not only because our visual system is highly sensitive to continuity — presenting the samples by themselves would

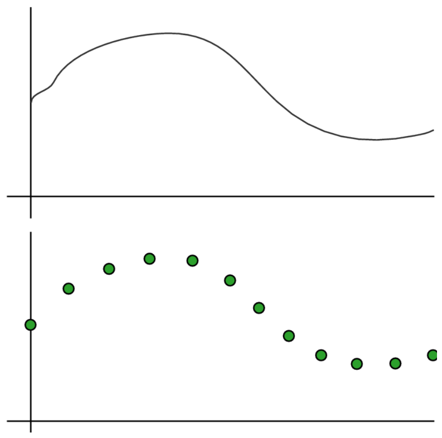

Figure 4.5. Sampling a function in regular intervals.

be inneffective — but because most visualization techniques require a continuous (and usually almost everywhere smooth) function.

Such a reconstruction is achieved by *introducing assumptions about the data* between the samples. There are naturally many different kinds of assumptions we can make, and each of these will lead to a different reconstruction scheme. Given the samples f_i from an original function $f(x)$, we will call the new reconstructed function $\tilde{f}(x)$. If we require that $\tilde{f}(x) = f(x)$ at the f_i sample points, we call this an *interpolation* function. If we are willing to relax the restriction and settle for $\tilde{f}(x) \approx f(x)$, we call it an *approximation* function. We will start with interpolation schemes, moving from simpler to more complex ones, and then discuss how the notion of *convolution* shows that all these reconstruction schemes are all deeply connected.

The simplest reconstruction technique of all is known as *nearest-neighbor interpolation*. As the name suggests, we fill out the unknown values by simply picking the value of the closest sample available. This is very simple to implement and very efficient. However, $\tilde{f}(x)$ will never be continous, regardless of how densely we sample $f(x)$. This makes nearest-neighbor reconstruction useful only in limited situations.

The next step is to enforce function continuity. The simplest way we can expect a function to be continous and respect given function values is to set its derivative to be constant between sample points. In other words, between every sample pair, we want $\tilde{f}(x)$ to be linear between f_k and f_{k+1}. To find out an explicit formula for the linear segments, we simply start with a model of what we assume one function segment to obey, and fill it in with our known values.

When working out these calculations, it will be convenient to change the parameterization. We will create a new function where the known values are at

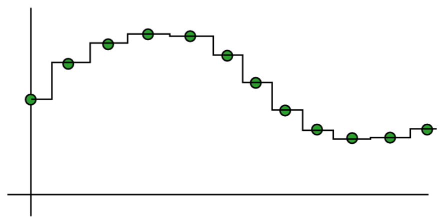

Figure 4.6. Nearest-neighbor reconstruction.

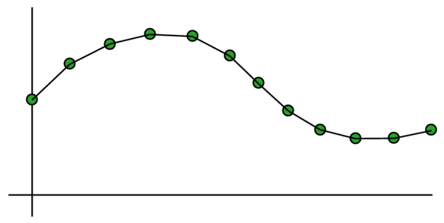

Figure 4.7. Linear interpolation.

$[0,1,2,\ldots]$, and simply transform this back when implementing the techniques in code. In our case, we only care about the samples directly to the left and right of the unknown value, so we start with $g(0)=k_0$, and $g(1)=k_1$. $g(x)$ is simply a coordinate transformation of $\tilde{f}$ that makes it more convenient to manipulation. Since we said the function should be linear, we make $g(x)=ax+b$, and now simply solve for the unknown parameters a and b, which is trivial:

$$\begin{aligned} g(x) &= ax+b \\ g(0) &= a.0+b=f_i \\ g(1) &= a+b=f_{i+1} \\ g(x) &= (f_{i+1}-f_i)a+f_i \end{aligned}$$

The resulting function is now continous, which is desirable for many scenarios. However, in some cases we need some level of smoothness in the derivatives that is not present in this reconstruction. We can naturally do better, and we show here a simple technique, known as *cubic interpolation*. The idea is exactly the same as linear interpolation, but instead of forcing the function to be piecewise linear, we force g to be a cubic: $g(x)=a+bx+cx^2+dx^3$. This function has two more parameters than the linear one, so we need to constrain our reconstruction somehow. Here, we will constrain $g(x)$ by prescribing values to its derivative. Specifically, we set $g'(0)$ based on the values of $g(1)$ and $g(-1)$, and set $g'(1)$

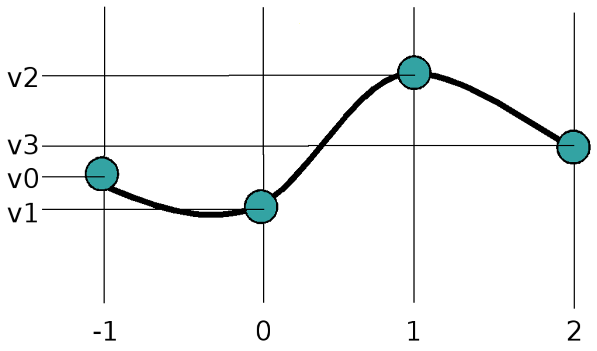

Figure 4.8. Cubic interpolation for a single segment.

based on $g(0)$ and $g(2)$. We will set them to be the slope of the line that would go between the points at -1 and 1, and 0 and 2 respectively.

Watch out for now for the notation mismatch on the figure. v0 $= f_{i-1}$, v1 $= f_i$, *etc.*

$$\begin{aligned}
g(x) &= a+bx+cx^2+dx^3 \\
g'(x) &= b+2cx+3dx^2 \\
g(0) &= f_i \\
g(1) &= f_{i+1} \\
g'(0) &= (f_{i+1}-f_{i-1})/2 \\
g'(1) &= (f_{i+2}-f_i)/2 \\
&\vdots \\
g(x) &= f_{i+1}+(f_{i+1}/2-f_{i-1}/2)x+ \\
&\quad (f_{i-1}-5/2f_i+2f_{i+1}-f_{i+2}/2)x^2+ \\
&\quad (-f_{i-1}/2+3/2f_i-3/2f_{i+1}+f_{i+2}/2)x^3
\end{aligned}$$

4.2.3 Accuracy

We would like to know whether $\tilde{f}$ is in any way similar to f away from its sample points. This is in general only possible if we have extra information about the global behavior of f. For example: if we know the largest magnitude of the derivatives (first or higher) of f, we can use repeated applications of the mean value theorem of integral calculus [Courant and John 90] to bound the difference between f and $\tilde{f}$ as a function of δ and the derivative magnitudes (one application for each derivative order).

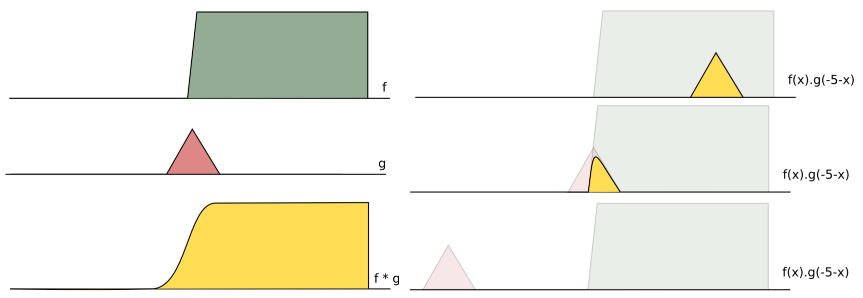

Figure 4.9. A convolution between two functions. The right column shows the product between f and several shifted instances of g. The shifted functions are showed ghosted out, and the product is shown in yellow. The integral of those products becomes the value of $f \star g$ in those three points. The functions and the convolution is shown in the left column.

4.2.4 Convolution

Even though this section is not strictly necessary to the rest of the text, the notion of *convolution* is so central to signal processing, that it is likely that you are going to encounter some instance of it in the scientific visualization literature. It is important, then, to have at least a certain amount of familiarity with it.

Convolution is a mathematical operator that takes two functions and produces a third function. Even though the definition of a convolution is completely symmetric, it is usually the case that one of the functions is called the *input*, and the other is called the *kernel*. The reason for this will become clear soon. The convolution between two functions f and g is denoted $f \star g$, and the formula for the convolution between two functions is:

$$(f \star g)(t) = \int_{-\infty}^{\infty} f(x)g(t-x)\,dx \tag{4.1}$$

Let us try to understand what the formula means. Every evaluation of $f \star g$ involves computing an entireintegral over the entire domain of f and g. Notice, however, that g is reflected about the origin, and then shifted by t. The two functions are then multiplied together and integrated. Figure 4.9 illustrates the idea.

As you can see, the convolution of f and g in Figure 4.9 looks like a smoothed version of f. In fact, if we think about the integral of the product in terms of Riemann sums, as long as $\int_{-\infty}^{\infty} g(x)\,dx = 1$, the convolution will replace each point of f with an analog of a weighted sum of its neighborhood, with the relative weights given by the values of g.

It turns out that most reconstruction techniques can be seen instances of the same procedure. As we will see, this idea of *convolving* one function with another will provide a framework under which we can create different reconstruction techniques much more conveniently. It also provides us with a systematic way to create reconstruction filters of progressively higher order. To get there, we first need to look at Dirac deltas.

The Dirac delta "function" The Dirac delta "function" is a very convenient mathematical tool to use when we need to mix discrete and continuous mathematics. In particular, it will be a way to represent discrete sets of function samples in a way that we can do calculus with it. The Dirac delta is defined by the following identity:

$$\int_{-\infty}^{\infty} f(x)\delta(x-\tau)\,dx = f(\tau) \tag{4.2}$$

The first thing to notice is that there is no explicit definition of $\delta(x)$. We will not worry about it too much, because we will never need to know the values of $\delta(x)$ directly. We will only use it through its defining identity. To get an intuition of what $\delta(x)$ looks like, however, let us examine the identity more carefully. $\delta(x)$ appears in a product with a function that is integrated over the entire real line. The result of that integral, however, is not a definite integral of f. Instead, it is simply $f(\tau)$.

For this identity to be true for any f, it must the case that for any $x \neq 0$, $\delta(x) = 0$. But then, if $\delta(0)$ is finite, the integral will be simply zero. For this reason, the Dirac delta is typically defined as a limit of gaussian functions:

Other definitions for $\delta(x)$ are also possible.

$$\delta(x) = g_k(x) = \lim_{k\to\infty} \sqrt{k/\pi}e^{-kx^2}$$

As k increases, the gaussian becomes narrower. In the limit, two important things happen: $\lim_{k\to\infty} g_k(x) = 0$ for $x \neq 0$, and $\lim_{k\to\infty} g_k(0) = \infty$. However, you should convince yourself that $\int_{-\infty}^{\infty} g_k(x)dx = 1$, for $k > 0$. *So a Dirac delta is, essentially, an infinitely narrow gaussian that integrates to one.* A Dirac delta is represented in a graph by a pointed arrow, whose position denotes shift, and length denotes scale, as Figure 4.10 illustrates.

Dirac deltas are extremely useful because we can represent a discrete set of samples as a set of shifted Dirac deltas. Each point sample will be a single Dirac delta centered at the position the sample was taken, and scaled by the intensity of the measurement, as shown in Figure 4.11. We will assume from now on that spike train deltas are spaced exactly 1 unit apart from each other.

This representation is sometimes known as a *spike train*.

Now we are ready to appreciate the power of convolution. You might have realized how similar Equations **??** and 4.2 are. The really important insight is that *we can use convolutions to compute weighted sums of Dirac delta heights*. In particular, Nearest-neighbor, linear, cubic and most reconstruction techniques *can all be expressed as convolutions with spike trains*. For example, we start defining a kernel as follows:

$$g_{\sqcap} = \begin{cases} 0, & |x| > 0.5 \\ 1, & \text{otherwise} \end{cases}$$

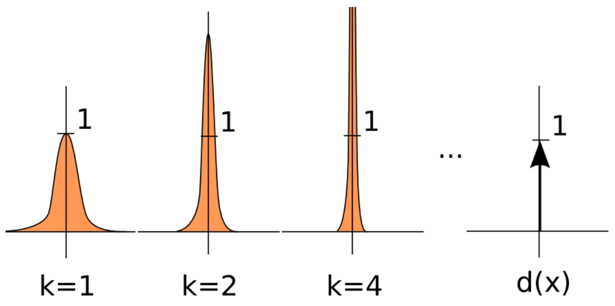

Figure 4.10. A Dirac delta is the limit of progressively narrower gaussians, and is denoted by an upward pointing arrow. Scaled Dirac deltas are shown with taller arrows.

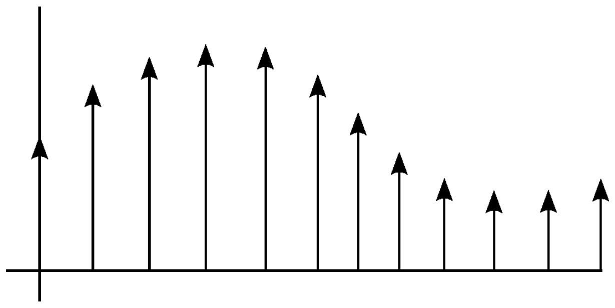

Figure 4.11. Representing a function as a set of Dirac deltas. (The irregular spacing is an artifact of the instructor’s lack of drawing ability)

If use the spike train representation and call it f, it should be easy to see that $f \star g_\sqcap$ is exactly a nearest-neighbor interpolation. This is because $g_\sqcap$ is only wide enough to “reach” a single delta spike at a time, and always equals to one, so the convolution will always exactly reproduce the height of the spike. We can define a different kernel:

$$g_\wedge = \begin{cases} 0, & |x| > 1 \\ 1 - |x|, & otherwise \end{cases}$$

FIXME: Add figures to all of this

as you might have guessed, $f \star g_\wedge$ is simply linear interpolation. This might seem uninteresting until you notice that $g_\wedge = g_\sqcap \star g_\sqcap$! So not only convolution gives us a unified way to analyze these reconstructions, it shows us that they are deeply connected to one another. It should come as no surprise that arbitrarily smooth reconstructions are possible, by convolving the reconstruction kernels with themselves repeatedly. These kernels are called *b-spline reconstruction ker-*

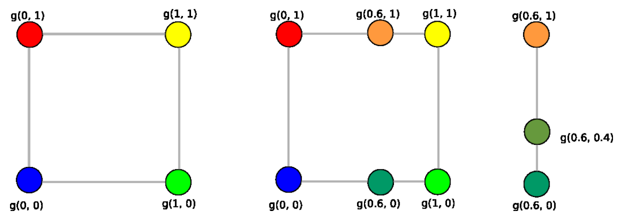

Figure 4.12. 2D separable reconstruction of $g(0.6, 0.4)$. Any 1-D reconstruction scheme can be used to reconstruct a n-dimensional field

.

nels, and are widely used in practice. The cubic interpolation we have seen before can also be recreated using a convolution with a kernel we call g_{CR}, the Catmull-Rom spline:

$$g_{\mathrm{CR}} = \begin{cases} 0, & |x| > 2 \\ -1/2|x|^3 + 5/2|x|^2 - 4|x| + 2, & 1 \leq |x| < 2 \\ 3/2|x|^3 - 5/2|x|^2 + 1, & 0 \leq |x| < 1 \end{cases}$$

4.2.5 From 1-D to n-D reconstruction

All the techniques we have discussed so far involve one-dimensional signal reconstruction. In scientific visualization, however, we are seldom interested in 1D signals. We need to extend our methods to deal with general n-dimensional reconstruction. Even though there are many different techniques to do so, we only present a very simple and general way, based on the notion of *separability*.

Sometimes, the phrase *tensor-product* reconstruction is used in place of separable reconstruction, not to be confused with the tensors in diffusion tensor imaging.

The idea in separable reconstruction is to realize that the n-dimensional problem can be broken down in several instances of 1-dimensional reconstruction. At each step i, we will have computed one-dimensional reconstructions for all the $0, 1, \cdots, i-1$ coordinates of the original point, and will use these to find a one-dimensional reconstruction using the i-th coordinate. Figure 4.12 illustrates two-dimensional separable reconstruction:

Separable reconstruction is very simple to understand and to implement. It works well enough in practice to be the predominant technique for n-dimensional reconstruction. There are issues, however, that should be noted, the most important to us being of performance. As the dimension of the field increases, the number of calls to the one-dimensional reconstruction function increases exponentially (can you see why?). This means that if a separable reconstruction is at the core of a scientific visualization algorithm (and as we will see later, this is often the case), the type of reconstruction might critically affect the performance of the entire visualization. In passing, we note that all the nice things that applied about 1D convolution translate directly to the n-dimensional separable convolution, the only difference being that the single integral is replaced by a multiple

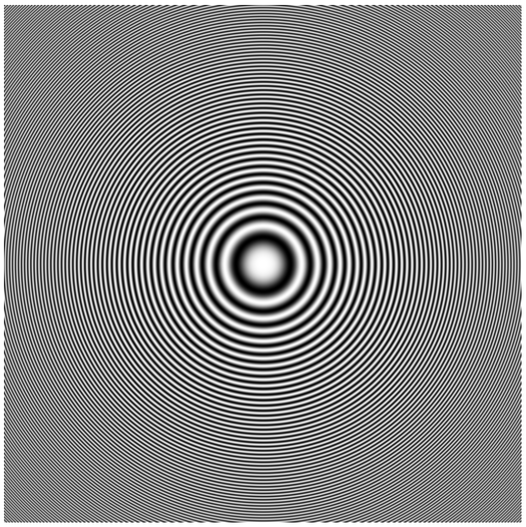

Figure 4.13. An example scalar field to be sampled.

integral.

4.2.6 Aliasing and Sampling

Interpolation and reconstruction kernels become very important when performing a fundamental operation on regular datasets: resampling. Often, the dataset we want to visualize is too large for the workstation we are using. Other times, some downstream algorithm takes too long to execute, so we want to replace the original data with a smaller version of it. Imagine we have a scalar field such as the one in Figure **??**. This scalar field is described by the following equations: $f(r) = \cos(r^2), x^2 + y^2 = r^2$. It is, in essence, a signal whose frequency increases as we move away from the origin.

If you are reading this electronically, zoom in as far as you can to see the field without artifacts.

If you are reading this electronically, zoom in and out to see even worse problems.

4.2.7 Picking a reconstruction technique

What are the tradeoffs between the different reconstruction techniques? As we just argued, there might be a significant performance overhead between different reconstruction techniques, simply because they're called so often. When, then, should we pick one technique over another. If the reconstruction needs to be analytically smooth, then there is no way to avoid a higher-order reconstruction like the cubic or higher-order b-splines we have seen before. On the other hand, if

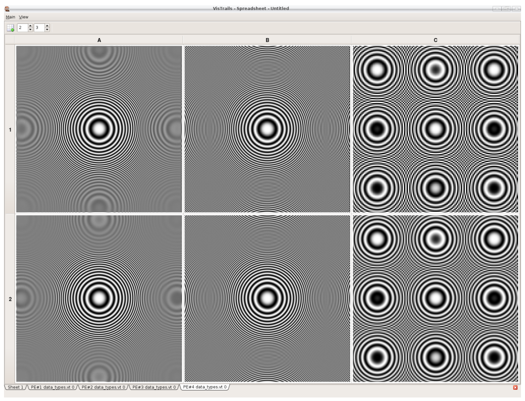

Figure 4.14. Sampling the scalar field in progressively coarser density shows jarring artifacts caused by *aliasing*.

visual quality is all we are interested in, linear or even nearest-neighbor interpolation might be sufficient, as can be seen on Figure 4.15.

There are no fixed set of rules regarding these decisions. It is important, then, as you are constructing your visualizations, to remember that they have an impact on the resulting quality and performance of your result.

4.3 Explicit vs. implicit representations

So far, we have only discusses *explicit* data representations. These are representations where there's an explicit representation of the domain and range of the object we are interested in. This typically involves finding an appropriate coordinate system and parameterization of the domain (which is the reason these representations are sometimes known as *parametric*).

Explicit representations have a distinct disadvantage when finding such a parameterization is complicated. For objects with complex topologies involving holes and handles, it is impossible to find a coordinate system that is continuous and covers the entire surface.

Implicit representations take a different perspective, and give up a parameterization for the simplicity of defining more complicated domains. In implicit representations, we store the domain as the *zero-set* of some other, explicit domain. For example, we can represent a unit circle parametrically as we have seen before, but we can also represent it by storing a function $f(x,y) = x^2 + y^2 - 1$ and claiming our domain is the set of points such as $f(x,y) = 0$. This set of points is typically referred to as the *implicit surface*, and the function is referred to as the *implicit equation*.

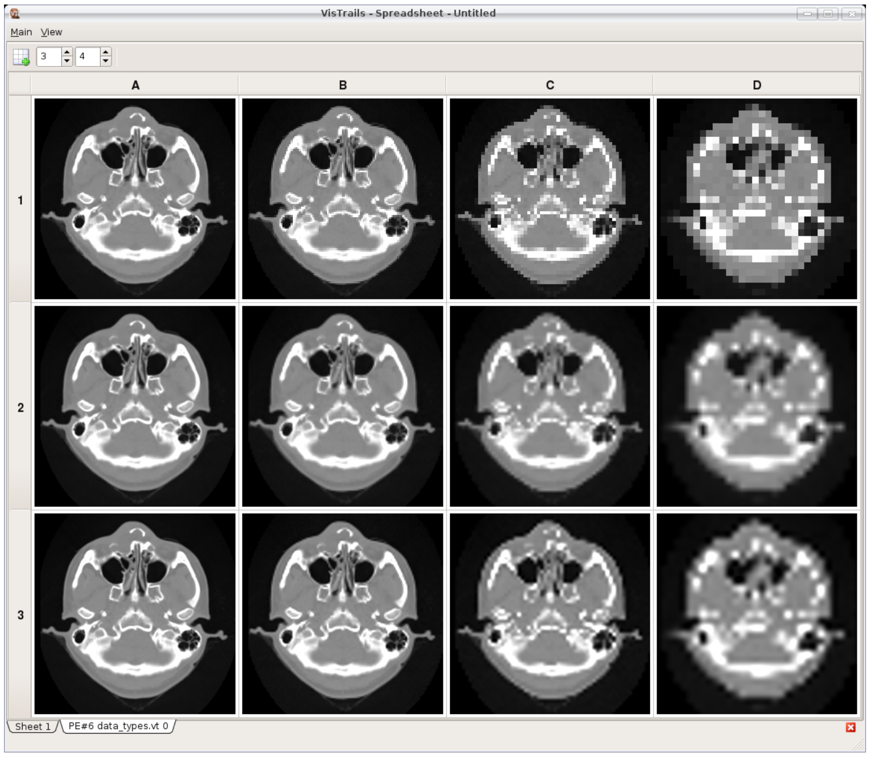

Figure 4.15. Exploring the space of interpolation techniques and sample spacing. The denser the sampling is, the less likely cubic interpolation will be noticeably better. The rows show, respectively, nearest-neighbor, linear and cubic reconstruction, while the columns show progressively coarser sampling.

The largest advantage of implicit representations is their flexibility. Implicit surfaces are largely indifferent to topology, since they are all defined on simple domains. Often, properties of the implicit surface can be inferred directly by the implicit equation. More importantly, it is possible to conveniently transform and process the implicit surface by manipulating the implicit equation. The biggest disadvantage, obviously, are the increased storage requirements. and not having an explicit parameterization.

4.4 Regular vs. Irregular Data

When modelling the physical world as a regularly spaced set of samples, we are implicitly assuming that every point in the world holds the same degree of importance as any other point. Also, since the processing in these regular data sets tend to be uniform across the grid, we are then also assuming equal interest in the approximation quality across the entire domain. In many cases, this assumption is inappropriate, and we use alternative representations. The main difference of these representations is they allow greater flexibility in the density of the samples we place in the physical world. In this section, we will examine these, starting with small changes to regular sampling and ending with models that have completely scattered data.

4.4.1 Semi-regular grids

Semi-regular grids (sometimes called *curvilinear meshes*) are the simplest extension to regular grids. If we think of the domain of regular grids as being a simple n-dimensional cartesian space, then we can imagine each point in a regular grid having a set of coordinates v_i associated with it. In the case of semi-regular grids, we sample the coordinate system uniformly, just like we did with regular data, but *the coordinate system does not regularly sample the domain*.

This is simplest to see with an example. Imagine we want to run a stress simulation on the profile of a hollow cylindrical bar. Figure **??** shows us what a regular sampling might look like. There clearly is a representation issue with the boundaries. Sampling the domain with enough density that the boundary becomes well-represented might mean we will use more data than necessary. The solution is to define the sampling on an appropriate coordinate system. In this case, polar coordinates offer us exactly such a mapping. The same domain can be represented much more accurately by circular ring sections, as seen in Figure **??**. In these figures, the samples are still represented a set of coordinates, but the coordinates are

$$f(\rho_i, \theta_j), \min < \rho \max$$

$$\rho(x,y) = \sqrt{x^2+y^2}$$

$$\theta(x,y) = \text{atan2}(y,x)$$

Let us examine the resulting configuration. Across concentric rings, each cell is markedly different from the other. Their difference in area is related directly to the Jacobian of the coordinate transformation $\rho(x,y), \theta(x,y)$. In this case, the larger the distance from the origin, the larger the distortion will be: the jacobian is exactly ρ. This can be either a disadvantage of the method, or might actually be put good use, if the jacobian corresponds roughly to the importance assigned to each cell. Most importantly, in parameter space, each of these ring sections has the same dimensions: the samples are equally spaced. This makes storing semi-regular grids as efficient as storing regular grids. Besides the coordinate transformations, no new information is necessary. The disadvantage is that it

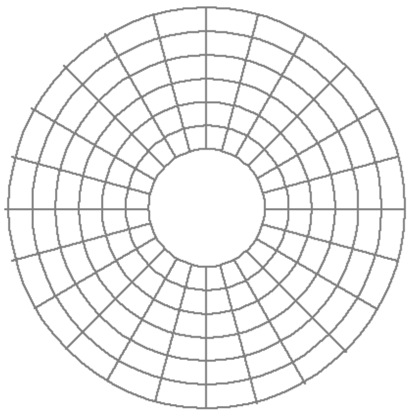

Figure 4.16. A semi-regular mesh samples the parameter space regularly, but uses a curved coordinate system.

becomes necessary to find a coordinate system that parameterizes the entire domain correctly, which is essentially impossible in more domains with complicated topology, with holes and handles.

4.4.2 Triangular and tetrahedral meshes

There are many situations where even semi-structured grids are not sufficiently flexible enough for the application. Fluid simulations typically need quite complex domains. The typical example is the simulation of air flowing around a wing profile. Aerodynamic engineers are particularly interested in the small-scale behavior just around the wing, but not far away from it. There is no natural parametrization of the domain that provides the wanted sample density, so the solution is to split it in a *triangular mesh*. For three dimensional fields, *tetrahedral meshes* offer the same advantages as triangular meshes do in 2D. Figure 4.17 shows two examples.

Triangular and tetrahedral meshes present a completely new issue to data modelling. Here, not only is density important, but the *shape* of the resulting elements becomes critical. Notice that for any set of samples, there are many different ways in which triangles can be generated. A complete discussion of the issues involved in picking a correct set of triangles given a set of point samples is beyond the scope of this book (there is an entire area of scientific computing called *meshing* that is involved in answering this and related questions). However, to give you an idea of how important these issues are, look at Figure 4.18, taken from [Shewchuk 02]. All three triangulations use the same amount of points, but the middle one has triangles very large angles. This is also the triangulation that incurs the largest amount of error in function values and especially in function

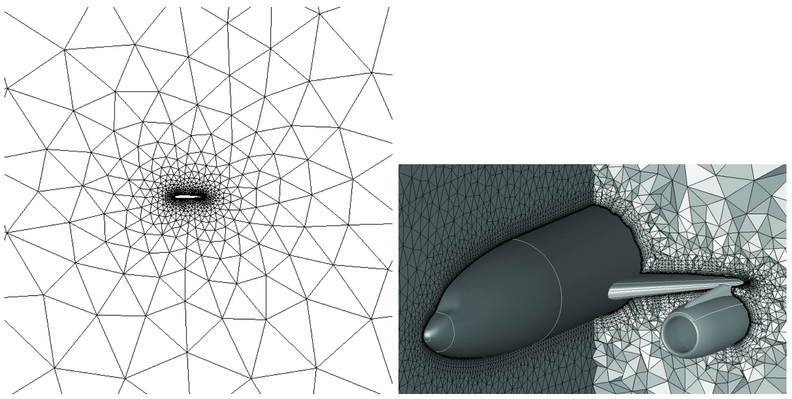

Figure 4.17. A triangle mesh representing a complex 2D domain, and a tetrahedral mesh representing a complex 3D domain.

gradients.

When discussing unstructured meshes, sometimes there is a distinction between *node centered data* and *cell centered data*. The distinction only refers to how the field is actually stored. If there is a function value stored at every vertex, and some sort of reconstruction is implied for the rest of the mesh, then we are dealing with node centered data. If each triangle or tetrahedron has enough data associated with it to completely determine the function inside its domain, then we call this a cell centered dataset.

4.4.3 Quadrilateral and hexahedral meshes

4.4.4 Scattered Points

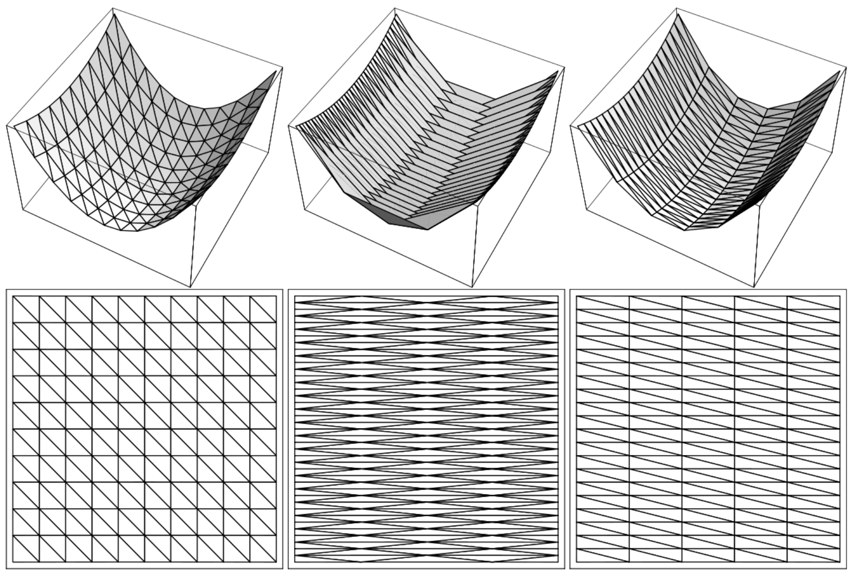

Figure 4.18. Example taken from Shewchuk [Shewchuk 02]. All three triangle meshes sample the exact same function. Notice, however, that the mesh in the middle has large errors, in particular in the approximated gradient. The middle mesh has with large angles, while the others don't.