



---

# Journal of Statistical Software

September 2008, Volume 28, Issue 1.

<http://www.jstatsoft.org/>

---

## Computing and Displaying Isosurfaces in R

Dai Feng  
University of Iowa

Luke Tierney  
University of Iowa

---

### Abstract

This paper presents R utilities for computing and displaying isosurfaces, or three-dimensional contour surfaces, from a three-dimensional array of function values. A version of the marching cubes algorithm that takes into account face and internal ambiguities is used to compute the isosurfaces. Vectorization is used to ensure adequate performance using only R code. Examples are presented showing contours of theoretical densities, density estimates, and medical imaging data. Rendering can use the **rgl** package or standard or **grid** graphics, and a set of tools for representing and rendering surfaces using standard or **grid** graphics is presented.

*Keywords:* marching cubes algorithm, density estimation, medical imaging, surface illumination, shading.

---

## 1. Introduction

Isosurfaces, or three-dimensional contours, are a very useful tool for visualizing volume data, such as data in medical imaging, meteorology, and geoscience. They are also useful for visualizing functions of three variables, such as fitted response surfaces, density estimates, or other density functions. The function `contour3d`, available in the R (R Development Core Team 2008) package `misc3d` (Feng and Tierney 2008), uses the marching cubes algorithm (Lorenson and Cline 1987) to compute a triangular mesh approximating the contour surface and renders this mesh using either the **rgl** (Adler and Murdoch 2008) package or standard or grid graphics. Several approaches are available for rendering multiple contours, including alpha blending for partial transparency and cutaway views.

The next section presents several examples illustrating the use of `contour3d`. The third section describes the particular version of the marching cubes algorithm used, and some computational issues. The fourth section presents utilities for representing surfaces and for rendering surfaces using standard or **grid** graphics. The final section presents some discussion

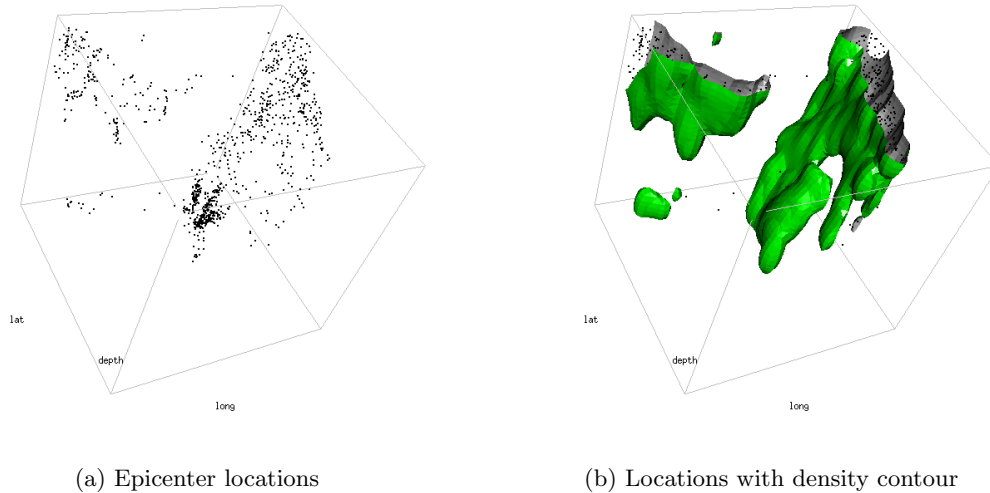


Figure 1: Locations of earthquake epicenters rendered using rgl.

and directions for future work.

## 2. Examples

The data set `quakes` included in the standard R distribution includes locations of epicenters of 1000 earthquakes recorded in a period since 1964 in a region near Fiji. Figure 1a shows a scatterplot of the locations. Figure 1b adds a contour of a 3D kernel density estimate, which helps to reveal the geometrical structure of the data. The scatterplot is created using

```
R> library("rgl")
R> points3d(quakes$long/22, quakes$lat/28, -quakes$depth/640, size = 2)
R> box3d(col = "gray")
R> title3d(xlab = "long", ylab = "lat", zlab = "depth")
```

The kernel density estimate is computed using the function `kde3d` in package `misc3d` and rendered using `contour3d` and the default `rgl` rendering engine:

```
R> de <- kde3d(quakes$long, quakes$lat, -quakes$depth, n = 40)
R> contour3d(de$d, level = exp(-12), x = de$x/22, y = de$y/28, z = de$z/640,
+           color = "green", color2 = "gray", add = TRUE)
```

The `d` component of the result returned by `kde3d` is a three-dimensional array of estimated density values. The argument `color` is the color used for the side of the surface facing lower function values; `color2` specifies the color for the side facing higher values, and defaults to the value of `color`.

The color arguments can be an R color specification or a function of three arguments, the  $x$ ,  $y$ , and  $z$  coordinates of the midpoints of the triangles. This can be used to color the triangles

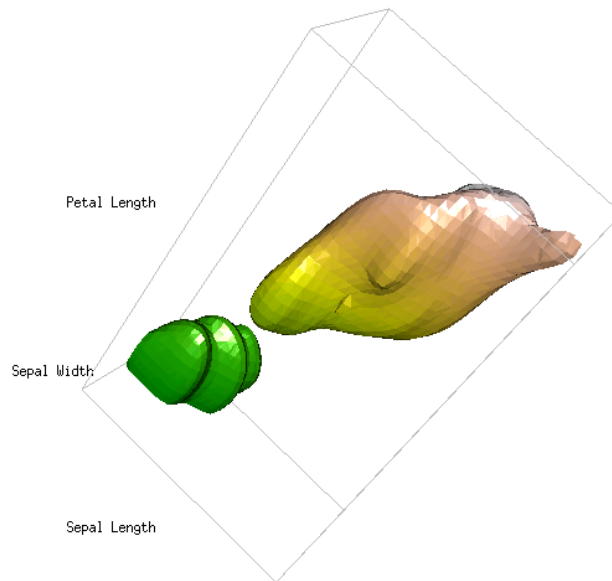


Figure 2: Density contour surface for variables `Sepal.Length`, `Sepal.Width`, and `Petal.Length` from the `iris` data set, with false color showing the levels of the fourth variable, `Petal.Width`, predicted by a loess fit.

individually, for example to use false color to encode additional information. Figure 2 shows a contour of a kernel density estimate of the marginal density of the variables `Sepal.Length`, `Sepal.Width`, and `Petal.Length` from the `iris` data (Anderson 1935). Color is used to encode the level of a loess fit of the fourth variable, `Petal.Width`, to the first three variables. This shows the positive correlation between `Petal.Length` and `Petal.Width`. The plot is created by evaluating the expressions

```
R> de <- kde3d(iris[,1], iris[,2], iris[,3], n = 40)
R> fit <- loess(Petal.Width ~ Sepal.Length + Sepal.Width + Petal.Length,
+             data = iris, control = loess.control(surface = "direct"))
R> fitCols <- function(x, y, z) {
+   d <- data.frame(Sepal.Length = x, Sepal.Width = y, Petal.Length = z)
+   p <- predict(fit, d)
+   k <- 32
+   terrain.colors(k)[cut(p, k, levels = FALSE)]
+ }
R> contour3d(de$d, 0.1, de$x, de$y, de$z, color = fitCols)
R> box3d(col = "gray")
R> title3d(xlab = "Sepal Length", ylab = "Sepal Width",
+         zlab = "Petal Length")
```

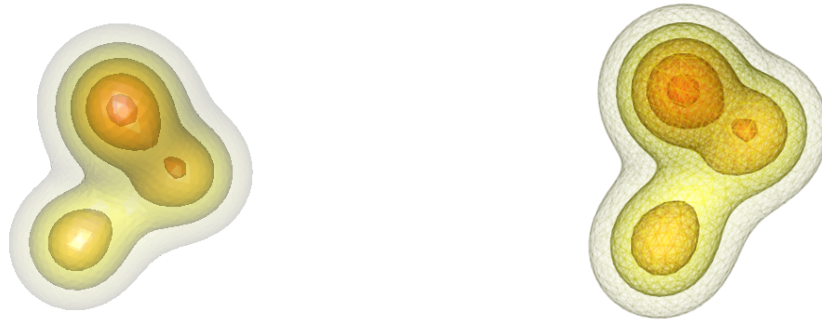
(a) Rendered with **rgl**(b) Rendered with **grid** graphics

Figure 3: Multiple isosurfaces of the density of a mixture of three tri-variate normal distributions rendered with partial transparency.

The faceting visible in Figure 2 can be reduced by using the argument `smooth`. For the **rgl** engine specifying `smooth = TRUE` results in computation of surface normal vectors at the vertices as renormalized averages of the surface normal vectors of the triangles that share the vertex; these are then passed on to the underlying **rgl** rendering function `triangles3d` for use in shading the triangles. Shading is described in more detail in Section 4.2. Shading is not used by default as it increases both computing time and memory usage and is not necessarily appropriate in all situations.

It is often useful to show multiple contour surfaces in a single plot. When surfaces are nested some means of revealing inner surfaces is needed. Two possible options are the use of partial transparency and cutaways. Figure 3 shows five nested contours of the mixture of three tri-variate normal densities defined by

```
R> nmix3 <- function(x, y, z) {
+   m <- 0.5
+   s <- 0.5
+   0.4 * dnorm(x, m, s) * dnorm(y, m, s) * dnorm(z, m, s) +
+   0.3 * dnorm(x, -m, s) * dnorm(y, -m, s) * dnorm(z, -m, s) +
+   0.3 * dnorm(x, m, s) * dnorm(y, -1.5 * m, s) * dnorm(z, m, s)
+ }
```

Multiple contours can be requested by providing a vector of more than one element as the `levels` argument to `contour3d`. Other arguments, such as `color`, are recycled as appropriate. Different levels of transparency, specified by the `alpha` argument, are used to produce the plot in 3a:

```

R> n <- 40
R> k <- 5
R> alo <- 0.1
R> ahi <- 0.5
R> cmap = heat.colors
R> lev <- seq(0.05, 0.2, length.out = k)
R> col <- rev(cmap(length(lev)))
R> al <- seq(alo, ahi, length.out = length(lev))
R> x <- seq(-2, 2, length.out = n)
R> contour3d(nmix3, lev, x = x, y = x, z = x, color = col, alpha = al)

```

In this case the first argument to `contour3d` is a vectorized function of three arguments, and the arguments `x`, `y`, and `z` that define the grid where the function is to be evaluated are required.

The plots shown so far have been rendered using `rgl`, with the views shown in the paper created with `snapshot3d`. `contour3d` also supports rendering in standard and grid graphics. This can be useful for incorporating contour surface plots in multiple plot displays or for adding a contour surface to a `persp` or `wireframe` plot. The rendering engine to use is specified by the `engine` argument; currently supported engines are `"rgl"`, the default, `"standard"`, and `"grid"`. Figure 3b is rendered using the `"grid"` engine in a PDF device that supports partial transparency by the code

```

R> alo <- 0.05
R> ahi <- 0.3
R> al <- seq(alo, ahi, length.out = length(lev))
R> pdf("normal-grid.pdf", version = "1.4", width = 4, height = 4)
R> contour3d(nmix3, lev, x = x, y = x, z = x, color = col,
+           alpha = al, engine = "grid")
R> dev.off()

```

Rendering in standard and grid graphics is done by filling polygons, and there currently does not appear to be a reliable way to avoid having polygon borders slightly visible when partial transparency is used. The border effect varies with the device and anti-alias settings.

Cutaways are another alternative for making inner contour surfaces visible. `contour3d` supports a `mask` argument for specifying which cells are to contribute to the contour surface. The argument can be a logical array with dimensions matching the data array argument, or a vectorized function returning logical values. Only cells for which the `mask` value is true at all eight vertices contribute to the contour surface. The `mask` argument can also specify separate masks for each level as a list of logical arrays or functions. Figure 4 shows the result obtained by the code

```

R> cmap <- rainbow
R> col <- rev(cmap(length(lev)))
R> m <- function(x,y,z) x > .25 | y < -.3
R> contour3d(nmix3, lev, x = x, y = x, z = x,
+           color = col, color2 = "lightgray",
+           mask = m, engine = "standard",
+           scale = FALSE, screen=list(z = 130, x = -80))

```

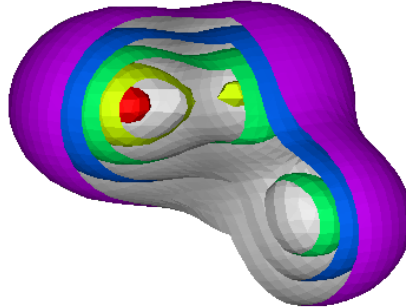


Figure 4: Isosurfaces of a normal mixture density rendered by standard graphics using a cutaway strategy to show the nested contours.

A `rainbow` color scale is used for the outside of the contours, with a neutral light gray color for the inside. The `scale = FALSE` argument specifies that the aspect ratio of the data should be retained. The viewpoint is adjusted using the `screen` argument, which specifies rotations in degrees around screen  $x$ ,  $y$ , and  $z$  axes; this interface is based on the interface for viewpoint specification used in the `lattice` functions `cloud` and `wireframe` (Sarkar 2008).

Volume data consisting of measurements on a regular three-dimensional grid arise in many areas, including engineering, geoscience, meteorology, and medical imaging. Isosurfaces of raw or smoothed data can be very useful for visualizing volume data. Figure 5 shows two contours of a CT scan of an engine block,<sup>1</sup> a standard example in the scientific visualization literature. After reading the three-dimensional data array into a variable `Engine`, the plot is produced by

```
R> contour3d(Engine, c(120, 200), color = c("lightblue", "red"),
+           alpha=c(0.1, 1))
```

The isosurface levels 120 and 200 were chosen by trial and error after examining a histogram of the intensity levels produced by `hist(Engine)`.

Figure 6 presents contours from two related data volumes from a study carried out at the Iowa Mental Health Clinical Research Center at the University of Iowa. The small red and yellow contours represent contours of mean differences in standardized blood flow between PET images for an active and rest period in a finger tapping experiment. These contours indicate areas of the brain that are activated in the finger tapping activity. To provide a spatial reference these contours are placed within a contour surface of a reference image of the brain constructed from normalized MR images of the subjects in the experiment. The

<sup>1</sup>The data set used was obtained from <http://www.sph.sc.edu/comd/rorden/engine.zip>.

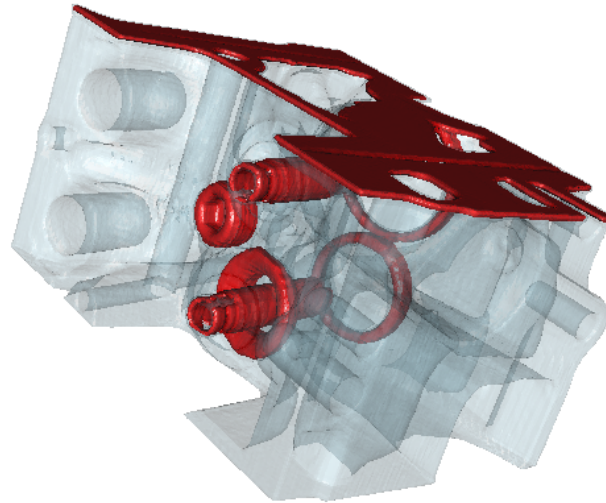


Figure 5: Two isosurfaces of a CT scan of an engine block.

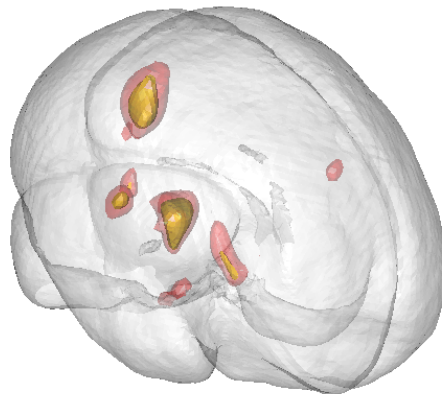


Figure 6: Isosurfaces of a brain and two intensity differences between two tasks in a PET experiment

data are stored in Analyze format, a common format used for medical imaging data, and can be read using functions provided by the **AnalyzeFMRI** package (Marchini and de Micheaux 2007). To construct this image we need to compute the two sets of contour surfaces separately and then render them as a single scene. For the **rgl** engine joint rendering is needed to ensure that transparency is handled correctly. Joint rendering is essential for the standard and grid engines since these draw the triangles making up a scene in back to front order.

The code for reading in the brain template and constructing the contours is given by

```
R> library("AnalyzeFMRI")
R> template <- f.read.analyze.volume("template.img")
R> template <- aperm(template,c(1,3,2,4))[, ,95:1,1]
R> brain <- contour3d(template, level = 10000, alpha = 0.3, draw = FALSE)
```

The argument `draw = FALSE` asks `contour3d` to compute and return the contour surface as a triangle mesh object without drawing it. The third line in the code is used to adjust the orientation of the image array and remove a fourth dimension that is not needed. The contour surfaces for the mean activation level are computed by

```
R> tm <- f.read.analyze.volume("tmap1-8.img")[, ,95:1]
R> brainmask <- template > 10000
R> activ <- contour3d(ifelse(brainmask, tm, 0),level = 4:5,
+                   color = c("red", "yellow"), alpha = c(0.3, 1),
+                   draw = FALSE)
```

The result in this case is a list of two triangle mesh structures representing the two contour surfaces requested. The basic rendering functions `drawScene` for standard and **grid** graphics and `drawScene.rgl` for **rgl** graphics accept either a single triangle mesh structure or a triangle mesh scene represented by a list of triangle mesh structures as argument. The three contour surfaces can therefore be rendered using the **rgl** engine with

```
R> drawScene.rgl(c(list(brain), activ))
```

The function `drawScene` with an appropriate `engine` argument can be used for rendering with standard or **grid** graphics.

### 3. Computing isosurfaces

This section presents the marching cubes algorithm and some computational considerations.

#### 3.1. The marching cubes algorithm

The marching cubes algorithm produces a triangular mesh approximation to the isosurface defined by  $F(x, y, z) = \alpha$  over a rectangular domain by a divide and conquer method starting from a set of values of the function  $F$  on a regular grid in the domain. The basic idea is that the grid divides the domain into cubes, and that how the surface intersects the cubes can be determined independently for each cube. For each cube, the first question is whether the cube intersects the isosurface or not. If function values at one or more of the vertices of a



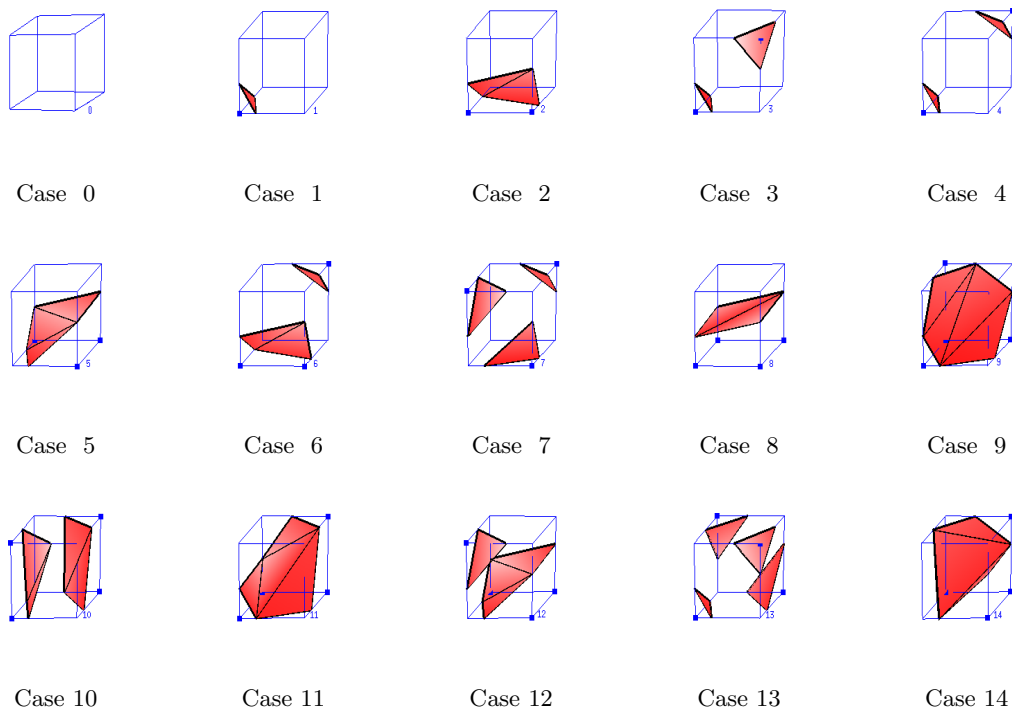


Figure 7: The original lookup table of the marching cubes algorithm

cube are above the target value  $\alpha$  and one or more function values are below, then the cube must contribute to the isosurface (for simplicity this discussion assumes all function values are strictly above or strictly below the target value  $\alpha$ ). After determining which edges of the cube are intersected by the isosurface, a triangular topological representation of the surface can be constructed.

Since there are eight vertices for each cube and the value of  $F(x, y, z) - \alpha$  at each vertex can be either negative or positive, there are  $2^8 = 256$  cases for each cube. Due to topological equivalence by rotation and switching between the positive and negative values, however, there are in total 15 distinct configurations that need to be considered; these configurations are shown in Figure 7, which was generated based on lookup tables used in `contour3d`. There are no triangles in configuration 0, since values of  $F(x, y, z) - \alpha$  at the vertices are either all positive or all negative. For configuration 1, all vertices, except the one at the front lower left corner, have the same signs while the front lower left corner has the opposite sign. Therefore, the isosurface separates the unique vertex from the others. The topological representation of the isosurface within a cube consists of a triangle or several triangles; the vertices of these triangles are the points at which the isosurface intersects the cube edges, and are determined by linear interpolation. The isosurface representation is accumulated by iterating, or marching, through all the cubes.

There has been extensive research on improving the quality of the topological representation produced by the marching cubes algorithm. [Nielson and Hamann \(1991\)](#) pointed out that there could be an ambiguity in the face of a cube when all four edges of the face are intersected

and the vertices on diagonal corners have the same signs, but the signs on the right diagonal are different than on the left. Face ambiguity is illustrated in Figure 8. From this figure, the



Figure 8: Illustration of face ambiguity

vertices on the right diagonal corners could be either separated (in case (a)) or non-separated (in case (b)). In this case, further calculation is needed to decide which pairs of intersections to connect. A remedy for face ambiguity can be based on the assumption that  $F$  is a bilinear function over a face,

$$F(s, t) = (1 - s, s) \begin{pmatrix} A & B \\ D & C \end{pmatrix} \begin{pmatrix} 1 - t \\ t \end{pmatrix}$$

with  $A$ ,  $B$ ,  $C$ , and  $D$  the function values at the vertices. It is easy to verify that the contour  $\{(s, t); F(s, t) = \alpha\}$  is a hyperbola. The asymptotes are  $\{(s, t); s = s_\alpha\}$  and  $\{(s, t); t = t_\alpha\}$ , where

$$s_\alpha = \frac{A - B}{A + C - B - D}$$

$$t_\alpha = \frac{A - D}{A + C - B - D}$$

So,

$$F(s_\alpha, t_\alpha) = \frac{AC - BD}{A + C - B - D}$$

The face ambiguity can be addressed by comparing the value at  $(s_\alpha, t_\alpha)$  with those at the vertices of the face. In Figure 8 for example, suppose the values of  $A - \alpha$  and  $C - \alpha$  are positive, and  $B - \alpha$  and  $D - \alpha$  are negative. If  $F(s_\alpha, t_\alpha) < \alpha$ ,  $A$  and  $C$  are separated as in Figure 9a; otherwise,  $A$  and  $C$  are connected as in Figure 9b.

In addition to face ambiguities, Chernyaev (1995) recognized that there are internal ambiguities, in terms of the representation of the trilinear interpolant in the interior of the cube. For example, in Figure 10, case 4 has two sub-cases. Two marked vertices could either be separated (case 4.1.1) or connected inside the cube (case 4.1.2). Furthermore, in order to make the triangular representation of the isosurface more conformable to the truth, an additional vertex might be needed (see case 7.3 in Figure 10 for example). The `contour3d` function mainly uses the algorithm suggested in Chernyaev (1995). The enlarged lookup table in Chernyaev (1995) is shown in Figure 10; this figure was generated based on lookup tables in `contour3d`. Note that there are still some sub-subcases which are not exhibited in the enlarged table. For example, for case 13.2 there are 6 sub-subcases, and only the one with positive vertices on the top face connected is shown.

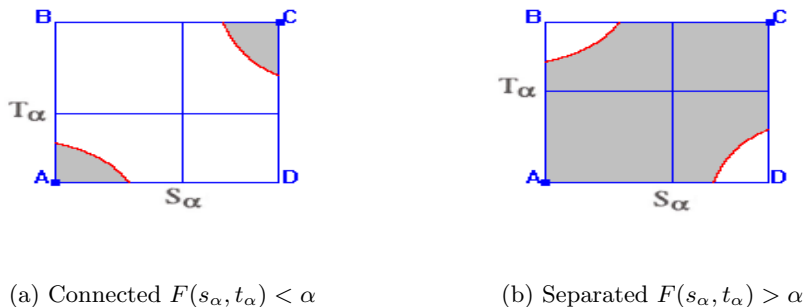


Figure 9: Resolving face ambiguity

### 3.2. Computational considerations

The core of the marching cubes algorithm is table-lookup. There are several tables in `contour3d` used to determine how an isosurface intersects each cube. For example, having 256 entries (each corresponding to one case), table `Faces` specifies which faces need to be checked to make further judgment on sub-cases. Table `Edges` shows, for each case, which edges are intersected by the isosurface. These tables are generated automatically based on basic configurations, their rotations, and switching of the positive and negative values.

In order to match each cube with a table entry, the execution flow could be serial, using a loop to iterate through the cubes one-by-one. This approach, however, is not very efficient in pure R code, although a compiler might be able to improve this. Besides, R facilitates vectorization very well by functions such as `ifelse`, `which` and so on. One important prerequisite for vectorization is that there is no dependency between successive inputs and outputs. In order to vectorize the marching cubes algorithm, each operation is executed on all cubes (or cubes with the same properties) simultaneously on the condition that there is no dependency among cubes. For example, the determination on cases of each cube and vertices of triangles (the bilinear interpolation) can be vectorized under careful coding. Computation of the table lookup index values can also be vectorized if care is taken. For example, for basic case 6 there are face and internal ambiguities. Two logical variables, `index1` and `index2`, are assigned for each ambiguity and a combined index is computed as `index = index1 + 2 * index2`.

## 4. Rendering surfaces in standard and grid graphics

Surfaces, such as three dimensional contour surfaces or surfaces representing functions of two variables, can be rendered by approximating the surfaces by a triangular mesh and passing the mesh on to a rendering function. The facilities provided by the `rgl` package are very well suited for interactive rendering and exploration, and can be used to generate snapshots as PNG images for inclusion in documents. At times it can also be useful to render triangle mesh surfaces using R's standard or `grid` graphics systems (Murrell 2005). Rendering in standard and `grid` graphics can be done by drawing the triangles in back to front order. The three-

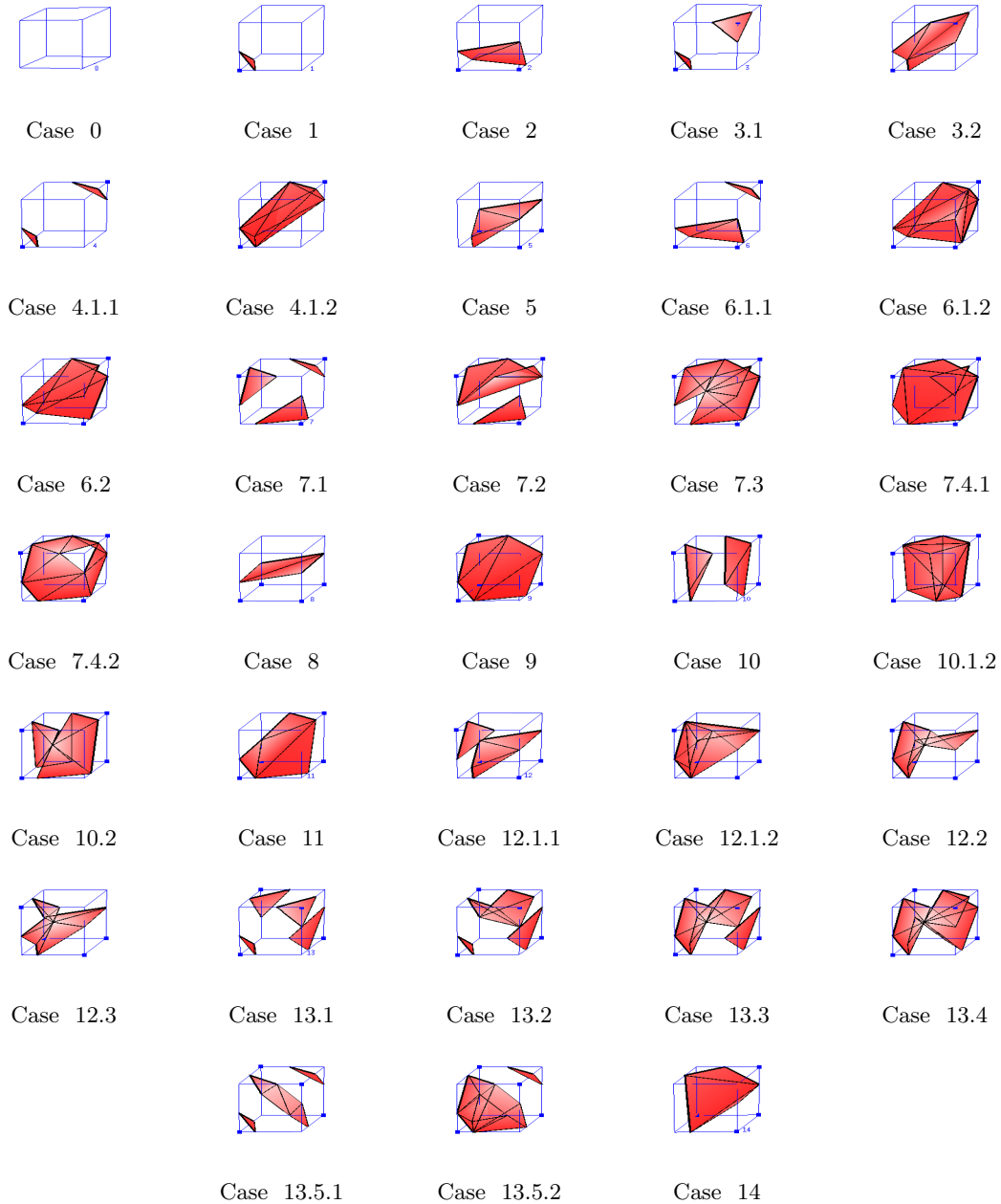


Figure 10: The lookup table of the marching cubes 33 algorithm

dimensional structure is brought out by adjusting the colors of the triangles according to a simple lighting model based on the direction of the triangle's surface normal vector relative to the position of the viewer and a lighting source. This section briefly describes the triangle data structure used in package **misc3d** and presents some details of the rendering method along with some illustrative examples.

#### 4.1. Triangular mesh surfaces

The triangle mesh data structure contains information representing the triangles, along with characteristics of the individual triangles and the surface as a whole that are used in rendering. The current representation is as a list object with the S3 class `Triangles3D`. For a mesh consisting of  $n$  triangles this structure currently includes components `v1`, `v2`, and `v3`, which are  $n \times 3$  matrices containing the coordinates of the vertices of the triangles. A more compact representation that takes into account the sharing of vertices is possible but not currently used.

Properties specified in the structure include `color` and `color2` for the color of the two sides of the triangles, `alpha` for the transparency level, `fill` indicating whether the triangles are to be filled, and `col.mesh` for the color of triangle edges. A final property, `smooth`, indicates whether shading is to be used to give the surface a smoother appearance. The color fields can contain a single color specification, a vector containing a separate color for each triangle, or a vectorized function used to compute the colors based on the coordinates of the triangle centers. `color` represents the color for the side for which the vertices in `v1`, `v2`, and `v3` appear in clockwise order. The `smooth` property is a non-negative integer value. For the standard and `grid` engines it specifies the level of shading to be used as described below in Section 4.2. For the `rgl` engine a positive value indicates that vertex normal vectors should be computed and passed to the `trianangles3d` rendering function.

Several functions are available for creating and manipulating triangle mesh objects. The functions `contour3d` and `parametric3d` create and optionally render contour surfaces and surfaces in three dimensions represented by a function of two parameters, respectively. The function `surfaceTriangles` creates a triangle mesh representation of a surface described by values over a rectangular grid. The constructor function `makeTriangles` creates a triangle mesh data structure from vertex specifications and property arguments. `updateTriangles` can be used to modify triangle properties, and `scaleTriangles` and `translateTriangles` to adjust the mesh itself.

#### 4.2. Rendering triangular mesh surfaces

Triangle mesh scenes are rendered by the function `drawScene`. This function performs the specified viewing transformation, computes colors based on a lighting model, possibly adding shading, performs a perspective transformation if requested, and passes the resulting modified scene on to the internal `renderScene` function. `renderScene` in turn merges the triangles into a single triangle mesh structure, optionally adds depth cuing, determines the  $z$  order of the triangles based on the triangle centers, and draws the triangles from back to front using the appropriate routine for drawing filled polygons using standard or `grid` graphics. The following subsections describe the lighting, shading, and depth cuing steps in more detail.

##### *Illumination*

Local illumination models (also called lighting or reflection models) provide a means of showing three dimensional structure in a two dimensional view by modeling the way in which light is reflected towards the viewer from a particular point on a surface. Simple models used in computer graphics usually consider two forms of light, ambient light with intensity  $I_a$  and light from one or more point light sources with intensity  $I_i$ , and two forms of reflection, diffuse

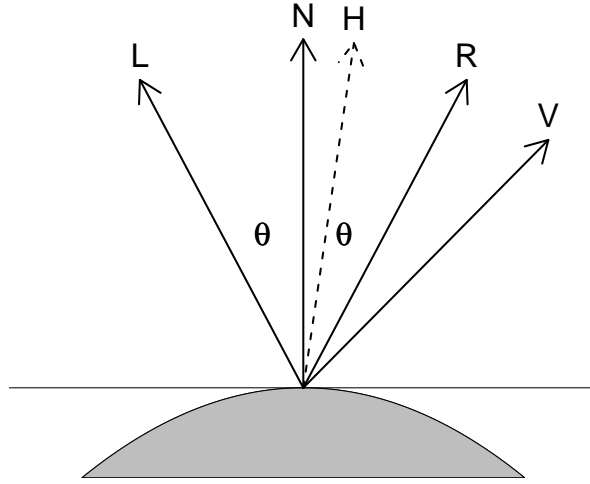


Figure 11: Vectors used in the reflection model.  $L$  is the direction to the light source,  $V$  is the direction to the viewer, and  $N$  is the surface normal.  $R$  is the reflection vector and  $H$  is the half-way vector proportional to  $(L + V)/2$ .

and specular. The light intensity seen by the viewer  $I_V$  is the sum of the intensities of an ambient component  $I_{Va}$ , a diffuse component  $I_{Vd}$ , and a specular component  $I_{Vs}$ . Separate intensities are used for the red, green, and blue channels but this is suppressed in the following discussion. The description given here is based mainly on [Foley, van Dam, Feiner, and Hughes \(1990, Section 16.1\)](#).

Ambient light represents a diffuse, non-directional source of light that illuminates all surfaces equally. The ambient component seen by the viewer is usually represented as

$$I_{Va} = I_a k_a O_a$$

where  $k_a$  is an ambient reflection coefficient associated with the object being rendered and  $O_a$  represents the object's ambient color.

Diffuse or Lambertian reflection reflects light from a point source equally in all directions away from the surface. The intensity of light from source  $i$  reflected towards the viewer is determined by the angle between the unit vector  $L_i$  in the direction of the light source and the unit normal vector  $N$  at the point of interest on the surface. The intensity of light from a single light source is

$$I_i k_d O_d \cos(\theta_{L_i, N}) = I_i k_d O_d (L_i \cdot N)$$

where  $k_d$  is the diffuse reflection coefficient associated with the object and  $O_d$  is the object's diffuse color. The intensity of diffuse light seen by the viewer is thus

$$I_{Vd} = \sum_i I_i k_d O_d (L_i \cdot N)$$

Specular reflection is the reflection of light off a shiny object. An ideal reflector reflects this light only in the direction of the reflection unit vector  $R$ , shown in Figure 11, and the color of

<b>ambient</b>	ambient reflection coefficient $k_a$
<b>diffuse</b>	diffuse reflection coefficient $k_d$
<b>specular</b>	specular reflection coefficient $k_s$
<b>exponent</b>	specular exponent $n$
<b>sr</b>	contribution of object color to specular color

Table 1: Components of a material structure

	metal	shiny	dull	default
<b>ambient</b>	0.45	0.36	0.3	0.3
<b>diffuse</b>	0.45	0.72	0.8	0.7
<b>specular</b>	1.50	1.08	0	0.1
<b>exponent</b>	25	20	10	10
<b>sr</b>	0.50	0	0	0

Table 2: Some pre-defined materials.

the reflected light matches the color of the light source. The Phong model (Bui-Tuong 1975) is a commonly used model for imperfect reflectors that represents the intensity of specularly reflected light seen by the viewer as a smooth function of the angle between the reflection vector  $R$  and a unit vector  $V$  in the direction of the viewer. This angle is twice the angle between the surface normal and the half-way vector  $H_i$  proportional to  $(L_i + V)/2$ , which is easier to compute. The specific version we have used represents the intensity of light from a single source specularly reflected towards the viewer as

$$I_i k_s O_s \cos^n(\theta_{H_i, N}) = I_i k_s O_s (H_i \cdot N)^n$$

where  $k_s$  is the specular reflection coefficient of the object,  $O_s$  is the object's specular color, and  $n$  is called the specular reflection exponent. For a perfect reflector the specular color is identical to the light color and the exponent is infinite. The total specular contribution is

$$\sum_i I_i k_s O_s (H_i \cdot N)^n$$

We use a simplified version of the model just described: Only a single white point light source is supported, and ambient light is white with the same intensity as the point light source. The ambient and diffuse object colors are assumed to be identical, and the specular object color is assumed to be a convex combination of the diffuse object color and white. The material characteristics are collected into a material structure with components listed in Table 1. Several materials are pre-defined, with characteristics based loosely on those found in MATLAB (The MathWorks, Inc. 2007); these are shown in Table 2. Rendering functions take a `material` argument that can be a character string naming a pre-defined material type or a list of the required components. Figure 12 shows a contour surface of a kernel density estimate from three variables of the iris data set rendered using the four pre-defined materials. The figure is created by

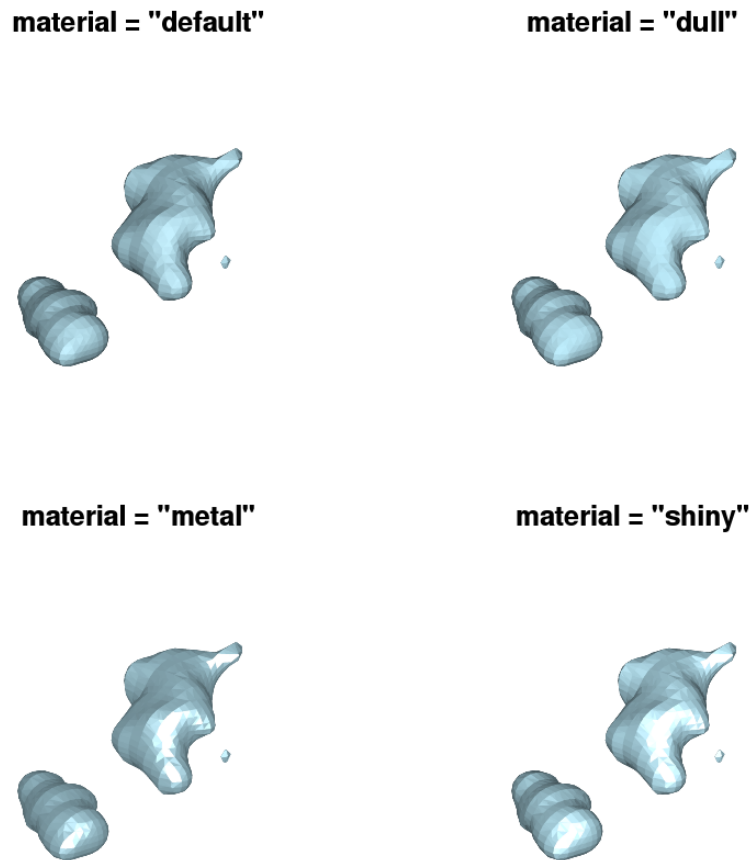


Figure 12: Contour surface of kernel density estimate for the first three variables in the iris data set rendered using the four standard material settings.

```
R> xlim <- c(4, 8)
R> ylim <- c(1, 5)
R> zlim <- c(0, 7)
R> de <- kde3d(iris[,1], iris[,2], iris[,3], n = 40,
+             lims = c(xlim, ylim, zlim))
R> opar <- par(mar = c(1, 1, 4, 1), mfrow = c(2, 2))
R> for (m in c("default", "dull", "metal", "shiny")) {
+   contour3d(de$d, 0.1, de$x, de$y, de$z, color = "lightblue",
+             engine = "standard", material = m)
+   title(paste('material = "', m, "'", sep = ""))
+ }
par(opar)
```



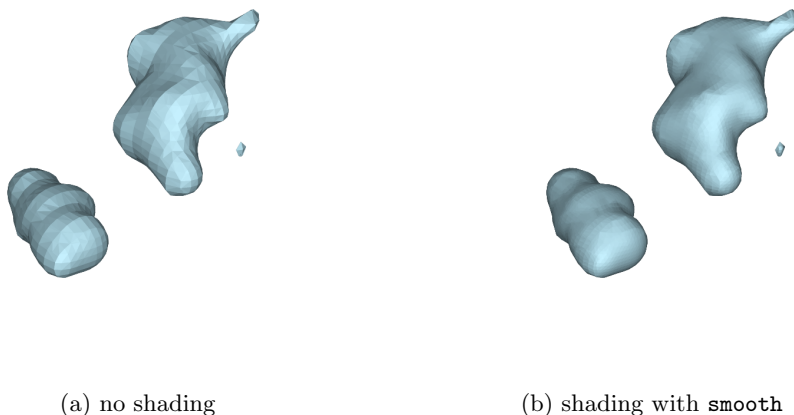


Figure 13: Contour surface of kernel density estimate for the first three variables in the iris data set rendered using no additional shading and using two iterations of shading.

### *Shading*

A triangular mesh is usually used as an approximation to a smooth surface. Rendering with an illumination model renders the approximate surface and clearly shows the facets of the approximation. Shading uses color variations within a facet to create a smoother representation. These color variations are computed based on surface normals. Suppose we have surface normals at each of the vertices of a facet. These may be available analytically or can be approximated by averaging the normals of the facets that share the vertex. One approach, known as Gouraud shading or intensity interpolation shading, computes colors for each vertex based on a lighting model and the vertex normals, and linearly interpolates colors across the facet. A second approach, Phong shading or normal vector interpolation shading, computes an interpolated normal vector for points within a facet and uses the interpolated normal vector to determine an appropriate color for the point.

Shading models are usually used at the pixel level, and often implemented in hardware. `rgl` uses this approach via the underlying `OpenGL` library. As a simple, though computationally costly, alternative for standard and `grid` graphics we can divide each triangle into four sub-triangles by splitting each edge in the middle, and apply either shading algorithm to the sub-triangles. This process can in principle be iterated several times. The `smooth` argument to `drawScene` specifies one plus the number of times to divide the triangles and uses the Phong shading model to compute appropriate colors. For `smooth = 1` there is no sub-division: the vertex normals are computed by averaging the triangle normals for triangles sharing the vertex, and a new surface normal for each triangle is then computed by interpolation of its vertex normals.

Figure 13 shows the density estimate contour surface for the iris data rendered with no shading and with `smooth = 2` corresponding to one level of subdivision. The figure is created with

```
R> opar <- par(mar = c(1, 1, 4, 1))
R> contour3d(de$d, 0.1, de$x, de$y, de$z, color = "lightblue",
+           engine = "standard", smooth = 0)
R> contour3d(de$d, 0.1, de$x, de$y, de$z, color = "lightblue",
+           engine = "standard", smooth = 2)
par(opar)
```

Aside from the higher cost in computing time and memory usage, using too high a level of `smooth` can cause the rendering quality to deteriorate as a result of artifacts due to the use of polygon filling for rendering the result. This may be exacerbated on devices that use anti-aliasing.

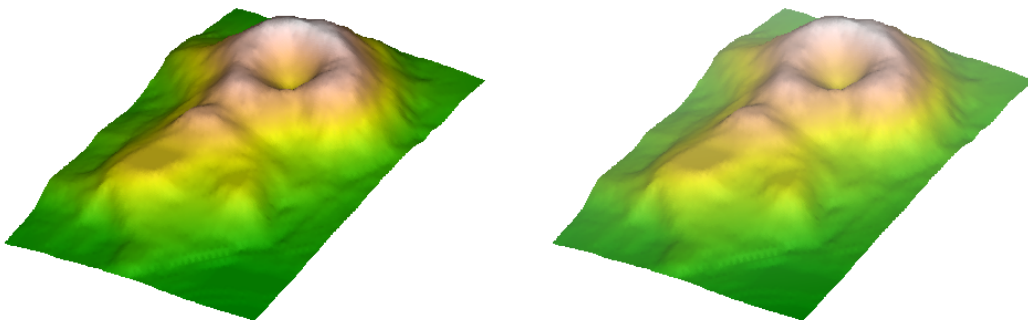
### *Atmospheric attenuation*

Simulated atmospheric attenuation can be used as a form of depth cuing to help indicate which parts of a scene are closer to the viewer and which are farther away. This approach blends the colors of more distant objects with the background color. The argument `depth` to the function `drawScene` specifies whether this form of depth cuing is to be used; if `depth` is non-zero then the rendering code computes the  $z$  values and maximal  $z$  value  $z_{\max}$  for the scene and sets

$$s = \frac{1 + \text{depth} \times z}{1 + z_{\max}}$$

Then the color intensities  $I$  are modified to  $sI + (1 - s)I_{\text{bg}}$  where  $I_{\text{bg}}$  represents the intensity of the background color.

Figure 14 illustrates depth cuing by simulated atmospheric attenuation using the elevation data for the Maunga Whau volcano included in the R distribution. Figure 14a uses no depth cuing and Figure 14b uses `depth=0.3`. The code to create the figures is



(a) no depth cuing

(b) with depth cuing

Figure 14: Surface plot of the Maunga Whau volcano with and without depth cuing and using `smooth = 3` level shading.

```

R> z <- 2 * volcano
R> x <- 10 * (1:nrow(z))
R> y <- 10 * (1:ncol(z))
R> vtri <- surfaceTriangles(x, y, z, color =
+           function(x, y, z) {
+             cols <- terrain.colors(diff(range(z)))
+             cols[z - min(z) + 1]
+           })
R> opar <- par(mar = rep(0, 4))
R> drawScene(updateTriangles(vtri, material = "default", smooth = 3),
+           screen = list(x = 40, y = -40, z = -135), scale = FALSE)
R> drawScene(updateTriangles(vtri, material = "default", smooth = 3),
+           screen = list(x = 40, y = -40, z = -135), scale = FALSE,
+           depth = 0.3)
R> par(opar)

```

### 4.3. Triangular mesh scenes and adding data points

The `drawScene` function can render a single triangular mesh or a scene consisting of a list of triangular mesh objects. This is useful for rendering multiple contour surfaces of a single function or for displaying contour surfaces of related functions or data sets in a single plot; this was used in constructing Figure 6. A somewhat frivolous example is shown in Figure 15.

Intersecting surfaces will be handled properly by the `rgl` engines but will appear ragged in the standard and `grid` engines due to the back to front drawing of triangles.

Rendering is restricted to triangular mesh objects in order to allow the use of vectorized computations at the R level. Quadrilaterals and other polynomial surfaces can be handled by

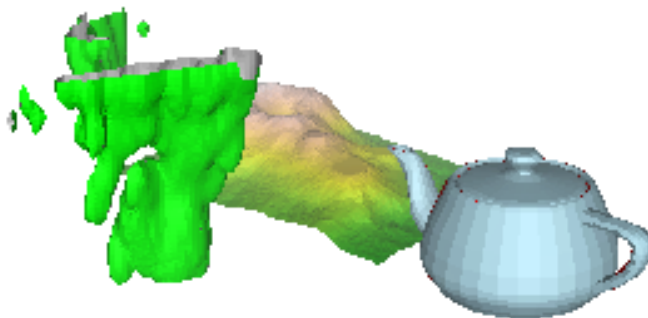


Figure 15: A scene combining several triangular mesh objects.

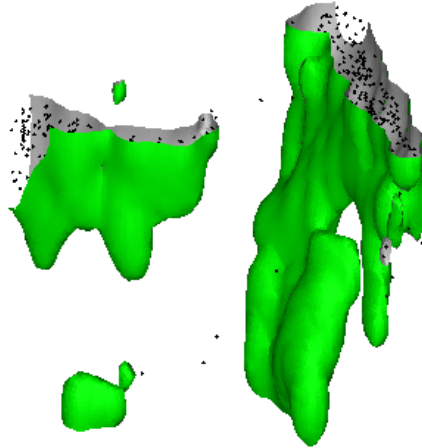


Figure 16: Earthquake epicenters and contour surface of kernel density estimate with points represented by small tetrahedra and `smooth = 2` shading of the contour surface.

decomposing them into triangles; this is the approach taken in `surfaceTriangles`. Adding data points to a plot, for example to a plot of a contour surface of a kernel density estimate as in Figure 1b, can be useful but does not fit directly into the triangular mesh framework. One possible solution is to represent each data point by a small tetrahedron. This can be done by the `pointsTetrahedra` function. An example is shown in Figure 16 and is created by the code

```
R> de <- kde3d(quakes$long, quakes$lat, -quakes$depth, n = 40)
R> v <- contour3d(de$d, exp(-12), de$x/22, de$y/28, de$z/640,
+               color = "green", color2 = "gray", draw = FALSE,
+               smooth = 2)
R> p <- pointsTetrahedra(quakes$long/22, quakes$lat/28, -quakes$depth/640,
+                       size = 0.005)
R> drawScene(list(v, p))
```

The `size` argument specifies the size of the point tetrahedra relative to the data ranges.

#### 4.4. Integrating with persp and wireframe plots

It can be useful to incorporate contour surfaces or other surfaces rendered by the methods described here within plots created by `persp` or the `lattice wireframe` function. As a simple illustration, these functions can be used to add axes to a contour surface plot. Figure 17 shows the results. These examples use a contour surface for three variables from the iris data computed by

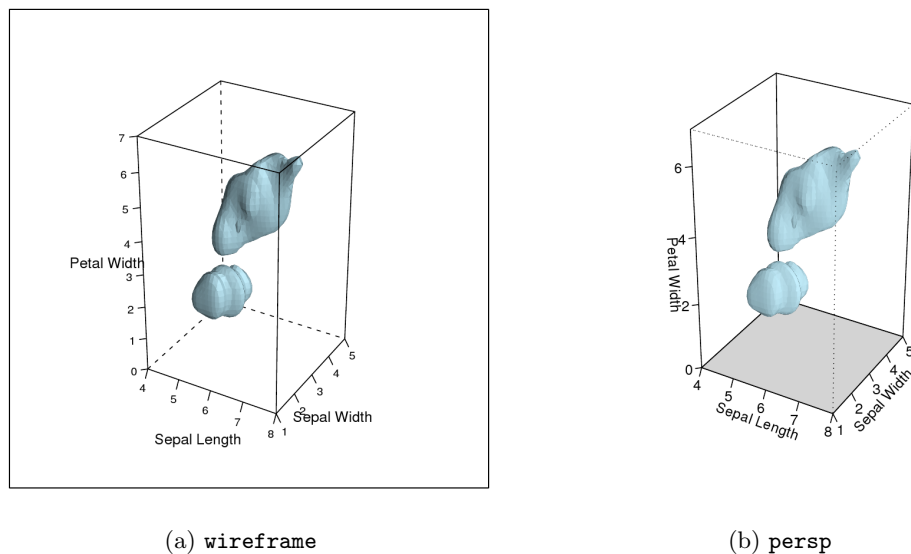


Figure 17: Using `wireframe` or `persp` to provide axes and labels for a contour surface of a kernel density estimate for the iris data.

```
R> xlim <- c(4, 8)
R> ylim <- c(1, 5)
R> zlim <- c(0, 7)
R> de <- kde3d(iris[,1], iris[,2], iris[,3], n = 40,
+             lims = c(xlim, ylim, zlim))
R> v <- contour3d(de$d, 0.1, de$x, de$y, de$z, color = "lightblue",
+                draw = FALSE)
```

Incorporating a contour surface in a wireframe plot is fairly straightforward and involves defining a custom `panel.3d.wireframe` function:

```
R> library("lattice")
R> w <- wireframe(matrix(zlim[1], 2, 2) ~ rep(xlim, 2) * rep(ylim, each = 2),
+                xlim = xlim, ylim = ylim, zlim = zlim,
+                aspect = c(diff(ylim) / diff(xlim), diff(zlim) / diff(xlim)),
+                xlab = "Sepal Length", ylab = "Sepal Width",
+                zlab = "Petal Width", scales = list(arrows = FALSE),
+                panel.3d.wireframe = function(x, y, z, rot.mat, distance,
+                xlim.scaled, ylim.scaled,
+                zlim.scaled, ...) {
+                scale <- c(diff(xlim.scaled) / diff(xlim),
+                diff(ylim.scaled) / diff(ylim),
+                diff(zlim.scaled) / diff(zlim))
+                shift <- c(mean(xlim.scaled) - mean(xlim) * scale[1],
+                mean(ylim.scaled) - mean(ylim) * scale[2],
+                mean(zlim.scaled) - mean(zlim) * scale[3])
+                P <- rbind(cbind(diag(scale), shift), c(0, 0, 0, 1))
```

```

+     rot.mat <- rot.mat %*% P
+     drawScene(v, screen = NULL, R.mat = rot.mat,
+             distance = distance, add = TRUE, scale = FALSE,
+             light = c(.5, 0, 1), engine = "grid")
+   })
R> print(w)

```

The panel function needs to compute the shifting and scaling that is done by `lattice` and incorporate these into the transformation matrix that is applied to the pre-computed contour. The call to `drawScene` with `add = TRUE` is then used to draw in the graphics viewport set up by `lattice` prior to calling the panel function.

Integration with `persp` is more involved because of the need to re-draw the bounding box. An initial plotting region can be set up with

```

R> M <- persp(xlim, ylim, matrix(zlim[1], 2, 2), theta = 30, phi = 30,
+           col = "lightgray", zlim = zlim, ticktype = "detailed",
+           scale = FALSE, d = 4, xlab = "Sepal Length",
+           ylab = "Sepal Width", zlab = "Petal Width")

```

This draws a light gray surface on the bottom of the region, along with axes and a bounding box. Evaluating

```

R> drawScene(v, screen = NULL, R.mat = t(M), add = TRUE, scale = FALSE,
+           light = c(.5, 0, 1))

```

adds the contour surface, but covers part of the front of the bounding box. The homogeneous coordinates transformation matrix used by `lattice` is the transpose of the matrix used by `persp`; we follow the `lattice` approach.

To reconstruct the bounding box we need a version of the R function `trans3d` that includes the  $z$  values of the transformed coordinates,

```

R> trans3dz <- function (x, y, z, pmat) {
+   tr <- cbind(x, y, z, 1) %*% pmat
+   list(x = tr[, 1]/tr[, 4], y = tr[, 2]/tr[, 4], z = tr[, 3]/tr[, 4])
+ }

```

and we use this to compute the transformed bounding box,

```

R> g <- as.matrix(expand.grid(x = 1:2, y = 1:2, z = 1:2))
R> b <- trans3dz(xlim[g[,1]], ylim[g[,2]], zlim[g[,3]], M)

```

and identify the front-most corner and its three neighbors:

```

R> fci <- which.max(b$z)
R> fc <- g[fci,]
R> coord2index <- function(coord)
+   as.integer((coord - 1) %*% c(1, 2, 4) + 1)
R> v1i <- coord2index(c(if (fc[1] == 1) 2 else 1, fc[2], fc[3]))
R> v2i <- coord2index(c(fc[1], if (fc[2] == 1) 2 else 1, fc[3]))
R> v3i <- coord2index(c(fc[1], fc[2], if (fc[3] == 1) 2 else 1))

```

Finally, the damaged bounding box is redrawn by

```
R> segments(b$x[fc1], b$y[fc1], b$x[v1i], b$y[v1i], lty = 3)
R> segments(b$x[fc1], b$y[fc1], b$x[v2i], b$y[v2i], lty = 3)
R> segments(b$x[fc1], b$y[fc1], b$x[v3i], b$y[v3i], lty = 3)
```

Perhaps in the future a simpler approach might become available.

## 5. Discussion

The contour surface computation and rendering functions presented in this paper could be enhanced in a number of ways. Adding support for bounding boxes and axes to `contour3d` might be useful; the `rgl` facilities used in some of the examples and the approach of integrating with `persp` or `wireframe` shown in Section 4.4 could serve as a starting point. Another useful addition might be a formula-based interface.

Several improvements in performance and memory use are also worth exploring. In cases where a function is provided, rather than a volume data array, it is possible to evaluate the function on the three-dimensional grid a small number of slices at a time and accumulate the contours. This can reduce memory use in some situations. Another option is to use *triangle decimation* (Schroeder, Zarge, and Lorensen 1992) to reduce the number of triangles needed to adequately represent a surface. It may also be useful to provide a limit on the number of triangles or the number of cells intersecting the surface to avoid high memory usage and paging in interactive settings when a user inadvertently requests a computation that would produce a contour surface consisting of an excessive number of triangles.

As shown by the examples in Section 2, isosurfaces or contour surfaces can be very useful in visualizing volume data and functions of three variables. The `misc3d` package also supports several other visualizations, includes an interactive tool for visualizing three axis-parallel slices of a data array provided by `slices3d`, and a visualization `image3d` based on rendering points or sprites with color and transparency determined by the data value. Other approaches, including cut planes, carpet plots, and ray casting methods may be added in the future.

## Acknowledgments

This work was supported in part by National Science Foundation grant DMS 06-04593. Some of the computations for this paper were performed on equipment funded by National Science Foundation grant DMS 06-18883. The authors would like to thank Ronald Pierson for making available the brain imaging data used in one of the examples.

## References

- Adler D, Murdoch D (2008). *rgl: 3D Visualization Device System (OpenGL)*. R package version 0.77, URL <http://CRAN.R-project.org/package=rgl>.
- Anderson E (1935). “The Irises of the Gaspe Peninsula.” *Bulletin of the American Iris Society*, **59**, 2–5.

- Bui-Tuong P (1975). “Illumination for Computer Generated Pictures.” *Communications of the ACM*, **18**(6), 311–317.
- Chernyaev EV (1995). “Marching Cubes 33: Construction of Topologically Correct Isosurfaces.” *Technical Report CN/95-17*, CERN, Institute for High Energy Physics.
- Feng D, Tierney L (2008). *misc3d: Miscellaneous 3D Plots*. R package version 0.6-2, URL <http://CRAN.R-project.org/package=misc3d>.
- Foley JD, van Dam A, Feiner SK, Hughes JF (1990). *Computer Graphics: Principles and Practice*. 2nd edition. Addison-Wesley, Reading, MA.
- Lorensen WE, Cline HE (1987). “Marching Cubes: A High Resolution 3D Surface Reconstruction Algorithm.” *Computer Graphics*, **21**(4), 163–169.
- Marchini JL, de Micheaux PL (2007). *AnalyzefMRI: Functions for Analysis of fMRI Datasets Stored in the ANALYZE Format*. R package version 1.1-7, URL <http://CRAN.R-project.org/package=AnalyzefMRI>.
- Murrell P (2005). *R Graphics*. Chapman & Hall/CRC, Boca Raton, FL. ISBN 1-584-88486-X.
- Nielson GM, Hamann B (1991). “The Asymptotic Decider: Resolving the Ambiguity in Marching Cubes.” In “VIS ’91: Proceedings of the 2nd conference on Visualization ’91,” pp. 83–91. IEEE Computer Society Press, Los Alamitos, CA. ISBN 0-8186-2245-8.
- R Development Core Team (2008). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL <http://www.R-project.org/>.
- Sarkar D (2008). *lattice: Multivariate Data Visualization with R*. Springer-Verlag, New York. ISBN 978-0-387-75968-5.
- Schroeder WJ, Zarge JA, Lorensen WE (1992). “Decimation of Triangle Meshes.” *Computer Graphics*, **26**(2), 65–70.
- The MathWorks, Inc (2007). *MATLAB – The Language of Technical Computing, Version 7.5*. The MathWorks, Inc., Natick, Massachusetts. URL <http://www.mathworks.com/products/matlab/>.

### Affiliation:

Dai Feng, Luke Tierney  
 Department of Statistics & Actuarial Science  
 University of Iowa  
 Iowa City, IA 52242, United States of America  
 E-mail: [dafeng@stat.uiowa.edu](mailto:dafeng@stat.uiowa.edu), [luke@stat.uiowa.edu](mailto:luke@stat.uiowa.edu)

---

*Journal of Statistical Software*  
 published by the American Statistical Association  
 Volume 28, Issue 1  
 September 2008

<http://www.jstatsoft.org/>  
<http://www.amstat.org/>  
 Submitted: 2008-05-28  
 Accepted: 2008-09-04

---