



CU/HPSC














Tutorial: Color in Scientific Visualization

**G. Domik, C. J. C. Schauble,
L. D. Fosdick, and E. R. Jessup**

This work has been supported by the National Science Foundation under an Educational Infrastructure grant, CDA-9017953 and developed as part of the High Performance Scientific Computing (HPSC) project at the University of Colorado at Boulder.

Copyright © 1999 by the HPSC Group of the University of Colorado

Table of Contents

-  [Introduction](#)
-  [Scientific Visualization](#)
-  [Human Visual Perception](#)
-  [Brightness](#)
-  [Color](#)
 -  [Color Models](#)
 -  [RGB Color Model](#)
 -  [HSV Color Model](#)
 -  [HLS Color Model](#)
 -  [Color Translation \(Lookup\) Tables](#)
 -  [Guidelines for using Color in Computer Graphics](#)
-  [Additional Exercises: Advection Data](#)
-  [Bibliography](#)

(Adapted from Chapter 2 of the HPSC Lab Manual: Computer Graphics and Visualization)

Introduction

A large amount of data is produced as a result of high performance scientific computing. The term, *visualization for scientific computing*, shortened to *scientific visualization*, was coined in 1986 and refers to the science or methodology of quickly and effectively displaying scientific data. The goal of scientific visualization is to enhance scientific productivity by utilizing human visual perception and computer graphics techniques.

Hence *computer graphics* has become an important part of scientific computing. A large number of software packages now exist to aid the scientist in developing graphical representations of his data. Such packages include MATLAB by [The MathWorks, Inc.](#), IDL (Interactive Data Systems) by [Research Systems, Inc.](#), AVS (Application Visualization System) by [Advanced Visual Systems, Inc.](#) and IRIS Explorer.

In order to use these graphics effectively, some understanding of visual perception is needed.

This tutorial serves as an introduction to some of these ideas; it is not meant to be a complete coverage of the subject. The examples and exercises are designed to acquaint you with some of the techniques and the problems involved in computer graphics. MATLAB and IDL are both able to handle colors, shadings and images; they are used in the following examples and exercises.

For further information on computer graphics, see [Foleyetal:1990]. A good source of information, specifically on the human eye and color vision, can be found in [GonzalezWoods:1992] and in [Travis:1991]. Concepts and experiments in visual thinking are well explained in [McKim:1980].

Scientific Visualization

It is difficult for the human brain to make sense out of the large volume of numbers produced by a scientific computation. Numerical and statistical methods are useful for solving this problem. Visualization techniques are another approach for interpreting large data sets, providing insights that might be missed by statistical methods. The pictures they provide are a *vehicle for thinking* about the data.

The two sides of the human brain function differently. The left side of the brain is the one that helps with analytical calculations. This part of the brain is the verbal part; it is able to handle abstract symbols, like numbers. Most scientists use this part of the brain to analyze detailed information, for instance, the numbers from a large computation.

The right side of the brain is the one that emphasizes spatial, intuitive, or holistic thinking. It allows the scientist to view an entire complex situation. Graphical representations of data stimulate this part of the brain. Using this approach, a scientist is able to get an overall picture of the data; later he use a more analytic method to examine certain anomalies or patterns in the data.

As the volume of data accumulated from computations or from recorded measurements increases, it becomes more important that we be able to make sense out of such data quickly. Scientific visualization, using computer graphics, is one way to do this.

Human visual perception

The eye is made up of several parts. Light enters the eye through the *pupil*, is focused by the *lens*, and passes through to the *retina* on the back of the eye. The retina is covered with sensitive nerve cells called *photoreceptors*; these cells receive the light and pass the stimulus onto the brain. The center of the retina is called the *fovea*; this is also the center of your vision.

The photoreceptors are divided into two groups: *rods* and *cones*. There are about 75 to 150 million rods, but only about 5 to 8 million cones. The rods provide the ability to detect brightness, and the cones allow you to perceive color. The rods are more *light*-sensitive than the cones, but are not able to distinguish between colors. Most of the cones are in the center of your eye, near the fovea; while the rods are absent from the immediate area of the fovea, but extend on both sides of the back of the retina. This is why you can usually only distinguish brightness levels at the edges of your vision.

There are three types of cones; each type is sensitive to different wavelengths of light. The cones for long wavelengths perceive mainly the yellows and reds; medium wavelength cones receive mainly the yellows and greens; and the short wavelength cones are strongly sensitive to the blue and indigo colors.

These three types of cones are not uniformly distributed on the retina. There are about 3.5 million each long and medium wavelength cones and they are mostly in the middle of the retina. On the other hand, there are only about 1 million short wavelength cones; these are distributed over the retina but are more strongly concentrated on the sides of the retina. This means that it is easier to focus on red, yellow, or green objects than on blue ones.

Brightness

The following examples and exercises are designed to further explore brightness as one aspect of human visual perception. [The striped image](#)

Example [[stripes1](#)]:

Using a graphical tool such as MATLAB or IDL, create an image containing sixteen vertical stripes of grey of increasing intensity. The brightness levels of the stripes should increase linearly from left to right with black as the leftmost stripe and white as the rightmost stripe. The image should contain 256x256 elements or pixels.

Solution:

A MATLAB script to produce such an image is discussed in [stripes1.m](#). An IDL program for the same purpose is given in [stripes1.pro](#). The image to be displayed is shown in the figure on the left. [Both this GIF image and the one following showing the grid of stripes were produced using `xfig`.]

Creating an image to be displayed in a window on the screen is much like doing needlepoint. For each grid location, you must pick a particular shade or color. If you are doing needlepoint, you attach a strand of yarn of the chosen color at the given intersection; in constructing an image, you insert a numeric value representing the chosen shade or color into the given location within a two-dimensional array. After all the grid locations are filled, the final picture appears when the image (or needlepoint) is displayed.

The IDL solution just uses a single stripe per color while the MATLAB version sets one stripe above a matching stripe of the same color. In addition, the MATLAB solution has matrix `c` stretched to define the

colors of the final image.

Exercise [stripes2]:

Run the program `stripes1.m` under MATLAB or run `stripes.pro` under IDL. Study both the program and the results of running the program.

Create and run a new program `stripes2.m` or `stripes2.pro` to do the same thing as `stripes1.m` or `stripes1.pro`, except for allowing the brightness levels of the stripes to increase logarithmically from left to right. The first and last stripes should match the brightness of the first and last stripes in the image created by the example above; this requires some scaling. Have the new program place the window for this image just above or below the window for the image created by the first program, so that you can see both images at once.

Hint: For MATLAB, consider the `logspace` function. For IDL, consider the `ALOG` or `ALOG10` function.

You may also wish to alter the number of stripes.

What is the difference, visually, between the two images? What, if anything, does this tell you about the perception of brightness by human eyes?

MATLAB: If you wish to remove the image window from your screen when you are done with this exercise, just use the MATLAB function, `close`. With no argument, it closes the current plot or image window. A sequence of `close` statements can be used to delete any number of accumulated display windows.



Example [squares1]:

Create a 512×256 window to hold two 256×256 images. The left half of the window should hold a square filled with a light grey value of 200 (out of 256); the right half should hold a square of grey value 50. Both halves should contain a centered, 100×100 , inner square with a grey value of 125 (out of 256).

Solution:

One solution to this example is the MATLAB script, [squares1.m](#); another is the IDL program [squares1.pro](#). These programs should be easy to follow. The color values for the two large squares are set first; then the inner square color values are written on top of them. Each element of an array is used to color one pixel of the final image.

Exercise [squares1]:

Run the MATLAB program, `squares1.m`. How does the perceived brightness of the two inner squares compare?

Exercise [squares1color]:

If you are using a color monitor, experiment with different colorings for `squares1.m` or `squares.pro`. For instance in MATLAB, replace the line

```
colormap(gray(256))
```

with

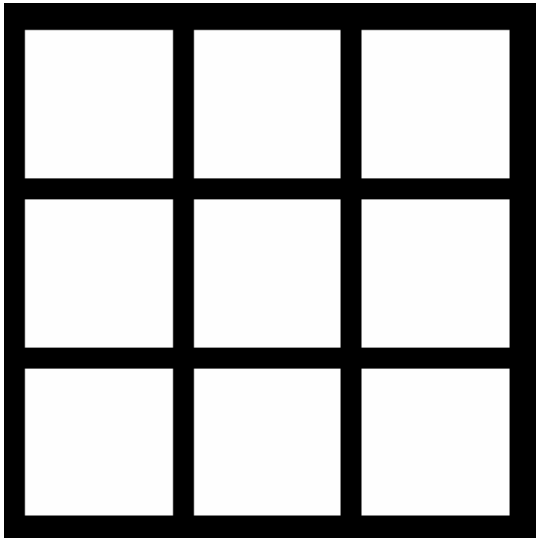
```
colormap(hsv(256))
```

To see what other colorings can be used in MATLAB, type

```
help color
```

In IDL, try different values for the parameter of the `LOADCT` command.

A more complete discussion on color maps is given later in this document.



Exercise [HermansGrid]:

The figure on the left displays the basic outline of Herman's grid. Create Herman's grid by writing a MATLAB script or IDL program to draw four white horizontal bars and four white vertical bars (each ten pixels wide) on a 265x265 black image. The image should be the reverse of that in the figure on the left.

When the image appears on the screen, stare at it for ten seconds. Do you see an illusion of grey spots at the grid intersections? When you try to focus on one intersection, the spot in that intersection should disappear.

Color

Color is an important part of graphics, especially with the increasing use of color monitors and color printers. Often color can be used to emphasize portions of your graphic productions.

The following sections discuss three different methods for modeling color, the use of color translation (or color lookup) tables for incorporating a specific set of colors into your graphics, and a list of guidelines for adding color to your graphics application.

Color models

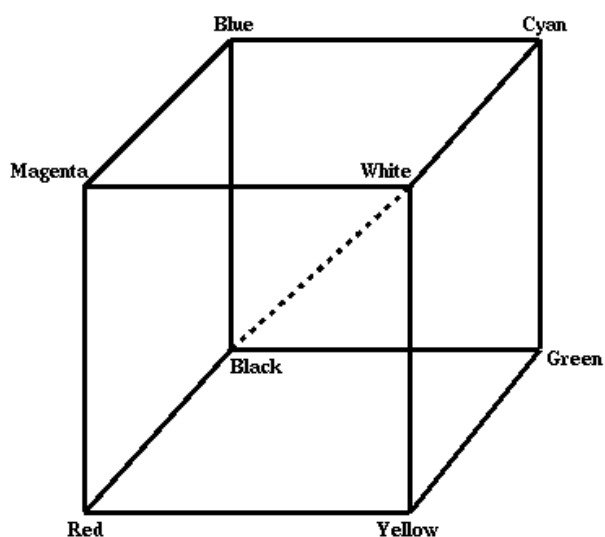
The term *primary colors* refers to three colors that are sufficient to create a gamut of colors -- the colors we expect to see in nature. An example of primary colors are red, green, and blue. The sum of all three of the primary colors -- that is, the mixture of all three colors in equal parts -- should form the color white. The absence of all three colors creates black.

Complementary colors are two colors that, when mixed, become the color white. Blue and yellow are two such colors.

Color monitors use this idea to create colored images. They use three different types of phosphor prisms at each pixel point on the screen; each type produces one of the three primary colors. The primary colors used by color television monitors (on which computer monitors were originally based) are *red*, *green*, and *blue*. The complements of these three colors are *cyan*, *magenta*, and *yellow*, respectively. Most discussions of color in computer graphics are based on these six colors in some way.

The term *brightness* describes the intensity of a color on a scale from 0.0 (dark) to 1.0 (bright) and is sometimes referred to as *value*. A black-and-white monitor can only show intensity, using 0 for black and 1 for white, with all the possible greys lying between. *Hue* is the term used to distinguish the dominant color as perceived by an observer and is related to its dominant wavelength. *Saturation* or *chroma* defines the purity of the color; for instance, pure blue is a more vivid color than pale blue (blue combined with white).

A number of color *models* have been developed. These attempt to represent a gamut of colors, based on a set of primary colors, in a three-dimensional space. Each point in that space depicts a particular color hue; some models also incorporate brightness and saturation.



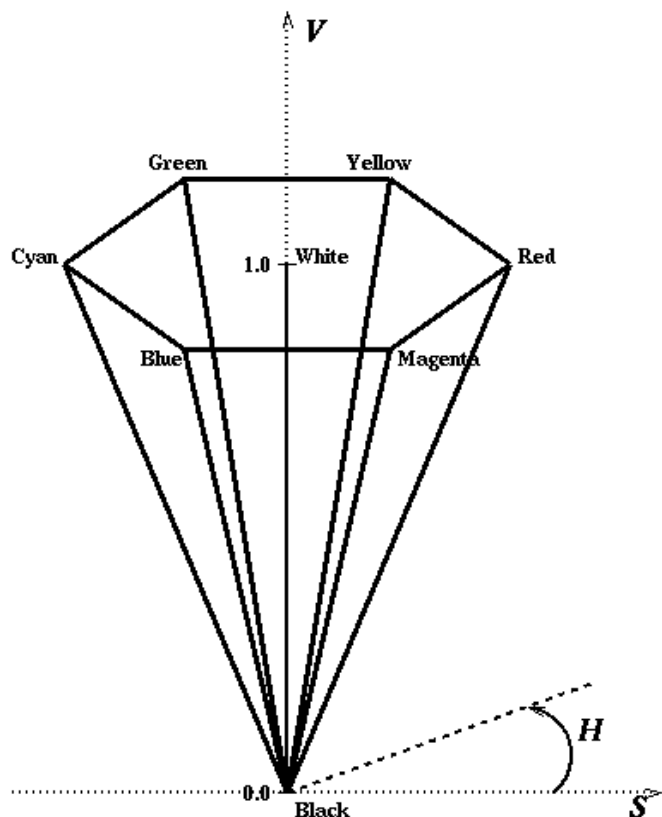
The RGB color model

The RGB (**R**ed, **G**reen, **B**lue) color model is shown in the figure on the left. This is built on a cube with Cartesian coordinates. Each dimension of the cube represents a primary color. Each point within the cube represents a particular hue; the coordinates of that point show the contributions of each primary color toward the given color.

The first coordinate represents the amount of red present in the hue; the second coordinate represents green; and the third coordinate refers to the amount of blue. Each coordinate must have a value between 0 and 255 for a point to be on or within the cube. Hence, pure red has the coordinate (255, 0, 0); green is located at (0, 255, 0); and blue is at (0, 0, 255). Thus, yellow is at location (255, 255, 0), and since orange is between red and yellow, its location on this cube is (255, 127, 0). The diagonal, marked as a dashed line between the colors *black* (0, 0, 0) and *white* (255, 255, 255), provides the various shades of grey. This model thus has the capability of representing 256^3 or more than sixteen million colors.

We should remind you that MATLAB only deals with double precision values. For this reason, the values in its color tables are not bytes, but real numbers between 0.0 and 1.0. When considering the values for the RGB color cube, MATLAB treats it as a unit cube. That is, 0.0 is equivalent to 0 (= 0/255), 0.2 is equivalent to 51 (= 51/255), and 1.0 is equivalent to 255 (= 255/255).

Both IDL and MATLAB use the RGB color model: for MATLAB, this is the basis for the `colormap` function and for IDL, this is the basis for the `LOADCT` and `TVCLT` commands. For more information, you may wish to type `help` in MATLAB or `?` in IDL. You may also refer to the IDL or MATLAB reference guides.



The HSV color model

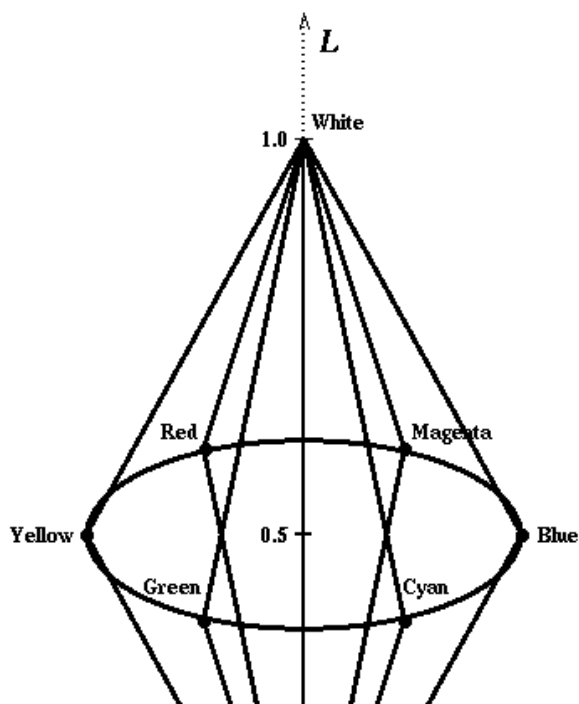
The HSV (Hue, Saturation, Value) color model, a cone, is shown in the figure on the left. This is one of the *perceptual color spaces* and was designed to mimic the way humans perceive color. The HSV color cone defines a color by hue, saturation, and value (brightness). The value or brightness of the color varies from zero to one along the axis, and the saturation of the color varies as the radial distance from the center axis. The hue is represented as an angle, with $H = 0$ degrees being red.

In the IDL version of this model, the other primary colors are 120 degrees apart.

The centerline from $v = 0.0$ to $v = 1.0$, between the colors *black* and *white*, provides all the shades of grey. Saturation values also vary from 0.0 to 1.0.

Since the parameters are all real numbers, the range of colors available should be infinite, but in fact, it is limited by the precision of the machine being used.

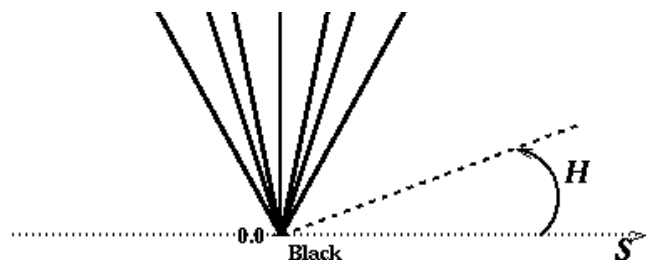
The default colormap for MATLAB is called `hsv`, but it is produced using the RGB model. Do not be confused.



The HLS color model

The HLS (Hue, Lightness, Saturation) color model is shown in the figure on the left. This is another *perceptual color space* and is similar to the HSV cone but with the primary colors located at $L = 0.5$ and with the colors of *black* and *white* acting as ends of the cones. Here the term, *lightness*, is related to brightness or intensity. Again, the L -axis from $L = 0.0$ to $L = 1.0$, between the colors *black* and *white*, provides the shades of grey. The values for saturation vary from 0.0 to 1.0.

As with the HSV model, the number of colors describable using this model is virtually infinite.



Color translation tables

Sample Color Table based on RGB
Cube

Pixel Value	Red Amount	Green Amount	Blue Amount
0	0	0	0
1	3	3	0
2	6	6	0
3	9	9	0
4	12	12	0
5	15	15	0
...
81	245	245	0
82	248	248	0
83	251	251	0
84	255	255	0
85	255	255	0
86	255	251	0
87	255	248	0
88	255	245	0
...
167	255	6	0
168	255	3	0
169	255	0	0
170	255	0	0
171	255	3	3
172	255	6	6
...
251	255	245	245
252	255	248	248
253	255	251	251
254	255	255	255

A *color translation table* or a *color lookup table (LUT)* associates a pixel value (0 to 255) to a color value. This color value is represented as a triple (i, j, k) ; this is much like the representation of a color using one of the color models. Once a desired color table or LUT has been set up, any image displayed on a color monitor automatically translates each pixel value by the LUT into a color that is then displayed at that pixel point. In MATLAB, an LUT is called a *colormap*; in IDL, it is a *color table*.

A sample color table is shown on the left. This table is based on the RGB color cube and goes gradually from black (0, 0, 0) to yellow (255, 255, 0) to red (255, 0, 0) to white (255, 255, 255). Thus, the pixel value (the subscript to the color table) for black is 0, for yellow is 84 and 85, for red is 169 and 170, and for white is 255. In MATLAB, the subscripts to the color table would run from 1 to 256.

Notice that not all the possible 256^3 RGB hues are represented, since there are only 256 entries in the table. For example, both blue (0, 0, 255) and green (0, 255, 0) are missing. This is due to the fact that a pixel may have only one of 256 8-bit values. This means that the user may choose which hues to put into his color table.

A color translation table based on the HSV or HLS color models would have entries with real numbers instead of integers. The first column would contain the number of degrees (0 to 360 degrees) needed to specify the hue. The second and third columns would contain values between 0.0 and 1.0.

Of course, a standard or default color table is usually provided automatically. Do a `help color` within MATLAB to see what color tables or *colormaps* it has predefined. For IDL help, type `?`.

255 255 255 255

Exercise [colors1]:

Write a MATLAB script named `colors1.m` or an IDL program named `colors1.pro` to create a 512×25 image and fill it with narrow vertical stripes of continuously increasing brightness values with black (0 in IDL or 1 in MATLAB) on the left edge and white (255 in IDL or 256 in MATLAB) on the right.

Load different color tables (including `gray`, `hsv`, `hot`, and `cool` in MATLAB or 0 through 10 in IDL) and examine the image in each. There should be a difference even on black and white monitors.

A user does not need to create his own color table unless he has some reason to do so. However, the next example shows how to set up a new specific color table in IDL or MATLAB, should it be necessary.

Example [colors2]:

Create a 512×25 image and fill it with narrow vertical stripes of continuously increasing brightness values with black (0) on the left edge and white (255) on the right.

Load an RGB LUT with values that create the following hues: black to yellow to red to white. Each hue should blend into the next.

Solution:

A MATLAB script to produce this image is discussed in [colors2.m](#). An IDL program for the same purpose is given in [colors2.pro](#). These are similar in many ways to the codes for the `stripes1` example, except that we are displaying a narrow strip of shades instead of a square of stripes. This example also calls for using a color table, based on the RGB color model. Conveniently, the table given in figure is the one needed for this example.

Comment: Some color monitors require the mouse pointer to be within the window displaying the image for the color to appear correctly.

Exercise [squares2]:

[This exercise requires a color monitor.]

As in the `squares1` example, write a MATLAB script or IDL program to create an image of two 256×256 squares, each containing a centered 100×100 inner square.

Create your own colormap to contain only four elements: the color representations for blue, orange, yellow, and one other color of your choice. Using this color map, color the large left square yellow and color the large right square blue. Set both inner squares to orange. Observe the color contrast. **Hint:** Review the section on the RGB color cube.

Call the program, `squares2.m` or `squares2.pro`.

Exercise [squares3]:

Repeat the `squares2` exercise using the HLS color space. Call the program `squares3.m` or `squares3.pro`. Which specification is easier: RGB or HLS? Why?

Guidelines for using color in computer graphics

A set of guidelines for the use of colors in computer graphics has been put together based on the manner most humans perceive colors. Some rules follow:

1. Use similar colors in data with similar characteristics; use distinct colors for distinct data.
2. Red is an action color; it calls attention to something. Blue is a background color; it usually shows status, meaning position or rank.
3. The eye is not as sensitive to blue as it is to other colors. First of all, most cones that receive short wavelengths (like blue and indigo) are on the edges, not in the center, of the retina. Secondly, the lens absorbs part of the wavelength light. Finally, the short wavelength cones that are distributed over the surface of the retina are far apart, due to their low number. For these reasons, small blue objects are hard to see and blue objects on a black background are almost impossible to see. The more one tries to focus on blue objects, the more they tend to disappear.
4. Don't have blue and red in your main object. Different colors require different types of focusing. Red objects are perceived by the cones for long wavelengths located in the center of the retina; blue objects are not. You may use red and green together; these two colors are complements of one another and are usually easy for the eye to distinguish. However, about 10% of the population has color-viewing deficiencies, and most of these people have difficulty differentiating between green and red. Blue and yellow are also complements of each other and hence good choices together.
5. If your picture uses only a few colors, you may use the complement of one of the main colors as the background. If there are several colors, use a grey or white background.
6. If you want people to concentrate on the main object in the center of the display, don't use a lot of different colors on the sides of your displays. If they concentrate on the middle, they will not be able to distinguish individual colors on the edges of the display.

Additional exercises: advection data

A given data file, `advect.dat`, represents a flow field at a particular time, imposed on a 50x50 grid. Each line contains the *x* and *y* coordinates of a point and the *z*-displacement of the flow at that point. The data for just one time step is given in this data set.

The `advect.dat` file is large. In compressed format it requires 21024 bytes; uncompressed, it takes 51500 bytes. It is available via ftp in [compressed format](#).

Each of the following examples and exercises refers to this set of data.

[More about the advection problem is discussed in chapters 13 through 18 of the HPSC Lab Manual.]

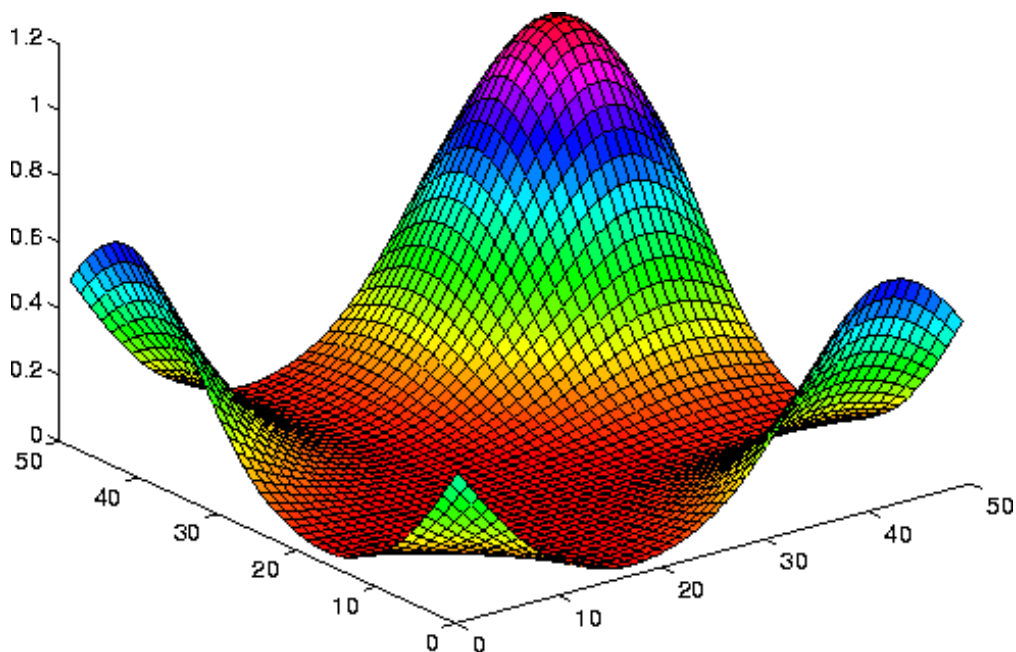
Example [advect1]:

Using MATLAB or IDL, produce mesh surface, shaded surface, and contour plots of the data in the file `advect.dat`. You need to use IDL or MATLAB input/output functions.

Solution:

The MATLAB script, [advect1.m](#), is a solution to this problem. The shaded surface plot below is one of the plots generated by this script. At the present time, there is no IDL solution for this example.

Since the data set may take some time to be read and manipulated, the program contains a few statements to inform the user of the progress through the program. Otherwise the wait might seem too long, and the user might think that there was a problem with the machine.



Exercise [advect1]:

Using MATLAB, run the `advect1.m` script. The `z` matrix is defined by the script and can be referred to after it completes. Try creating other 3-D plots of `Z`.

Exercise [advect1color]:

If you have a color monitor, try viewing the plots with different color tables. Just typing a `colormap`

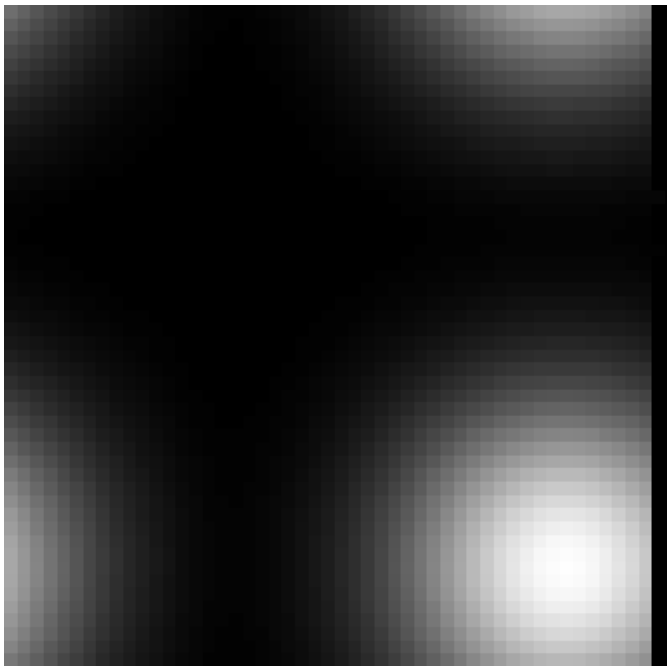
command should change the color in the current plot window. Which color table do you personally prefer for each of the plots? Can you provide a reason for these preferences?

Exercise [advect1IO]:

Study the MATLAB I/O functions and be sure you understand how they work for this example. If you are familiar with C, learn how these functions differ from the similarly-named C functions.

Explain each element used in the line of the `advect1.m` script that reads in data values:

```
[arow] = fscanf (advdat, '%g %g %g', [3 1]) ;
```



Exercise [advect2]:

Using IDL or MATLAB, create a color image of the displacements of all the points of this grid. The colors should be brightest for the extreme displacements; dark for small displacements. Call this program `advect2.pro` or `advect2.m`.

This should be an image, not a three-dimensional plot. But it should be reminiscent of the contour plot for this data. See the figure on the left for an example of this image.

If you have a color monitor, try viewing the image using different color tables. Which color table do you personally prefer for this image? Why?

Hint: Assign a color value to each element of the image according to the displacement at that point on the grid,

where the maximum displacement has a color value of 255 (256 in MATLAB) and the minimum displacement has a color of 0 (1 in MATLAB).

Also, to make each grid point be a 6x6 pixel square in the image, the window for the image needs to be six times the size of the grid.

Bibliography

[DEC:1990]: *DEC AVS User's Guide*, Digital Equipment Corporation (1990), Maynard, MA.

[Foleyetal:1990]: J. D. Foley, A. van Dam, S.K. Feiner, and J. F. Hughes, *Computer Graphics: Principles and Practice*, 2nd Edition, Systems Programming, Addison-Wesley Publishing Company (1990), New York, NY.

[**Fosdicketal:1996**]: L. D. Fosdick, E. R. Jessup, C. J. C. Schauble, and G. Domik, *An Introduction to High-Performance Scientific Computing*, The MIT Press (1996), Cambridge, MA, ISBN 0-262-06181-3.

[**GonzalezWoods:1992**]: *Digital Image Processing*, 2nd Edition, Addison-Wesley Publishing Company (1992), New York, NY.

[**MathWorks:1992a**]: *MATLAB: Reference Guide*, The MathWorks, Inc. (1992), Natick, MA.

[**MathWorks:1992b**]: *MATLAB: User's Guide for UNIX Workstations*, The MathWorks, Inc. (1992), Natick, MA.

[**McKim:1980**]: *Experiences in Visual Thinking*, 2nd Edition, PWS-Kent Publishing Company (1980), Boston, MA.

[**ResearchSystems:1993a**]: *IDL Reference Guide*, Research Systems, Inc. (1993), Boulder, CO.

[**ResearchSystems:1993b**]: *IDL User's Guide*, Research Systems, Inc. (1993), Boulder, CO.

[**Travis:1991**]: *Effective Color Display*, Academic Press, Inc. (1991), San Diego, CA.

[CSCI 4576/4586](#): Return to Home Page for CSCI 4576/4586

[CU CS Home](#): Return to CU Boulder Computer Science Dept Home Page

The HPSC Group: ERJessup jessup@cs.colorado.edu
CJCSchauble schauble@cs.colorado.edu
LDFosdick lloyd@cs.colorado.edu
Dept. of Computer Science,
University of Colorado,
Boulder, CO 80309-0430

CU/HPSC