Image Algorithms

"Image Algorithms"
http://people.sc.fsu.edu/~jburkardt/presentations/
asa_2011_images.pdf

.....

ISC4221C-01:
Algorithms for Science Applications II

.....

John Burkardt
Department of Scientific Computing
Florida State University

Haleh Ashki Fall Semester 2013

Spring Semester 2011

Image Algorithms

- **Overview**
- Representing an Image
- The Portable Gray Map Format
- Image Histograms
- Contrast Stretching
- Contrast Enhancement
- Salt and Pepper Noise
- Gaussian Noise
- Edge Detection
- Components
- Projects

OVERVIEW: Images

To computational scientists, images are just data.

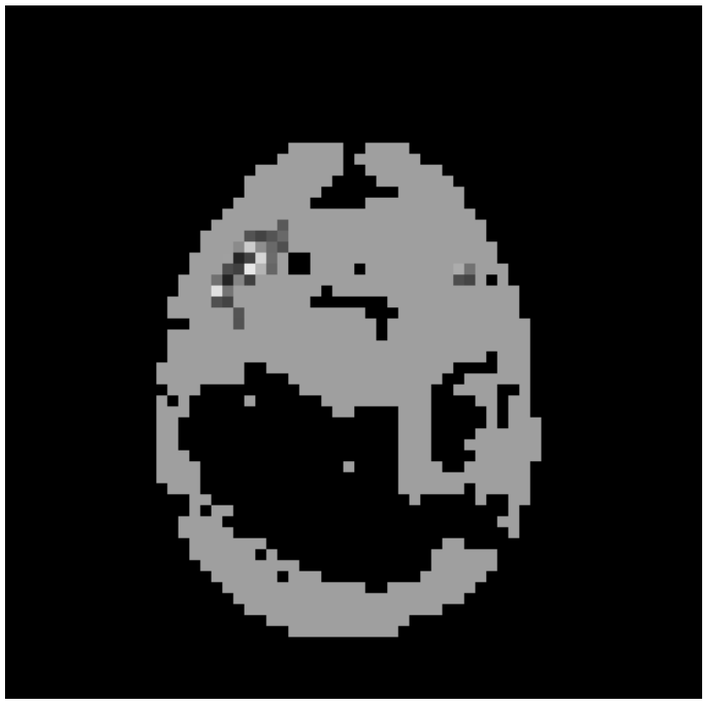
Images are created and stored digitally, so maps and portraits and X-rays all become numeric data.

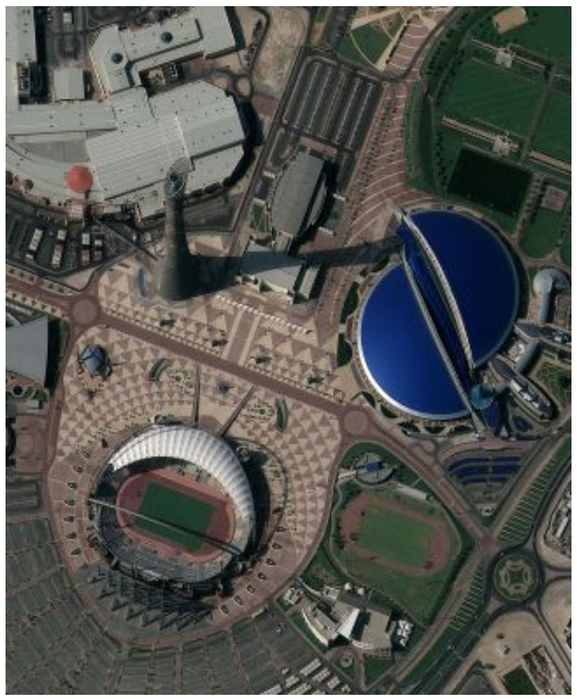
Image processing involves manipulating the image data to:

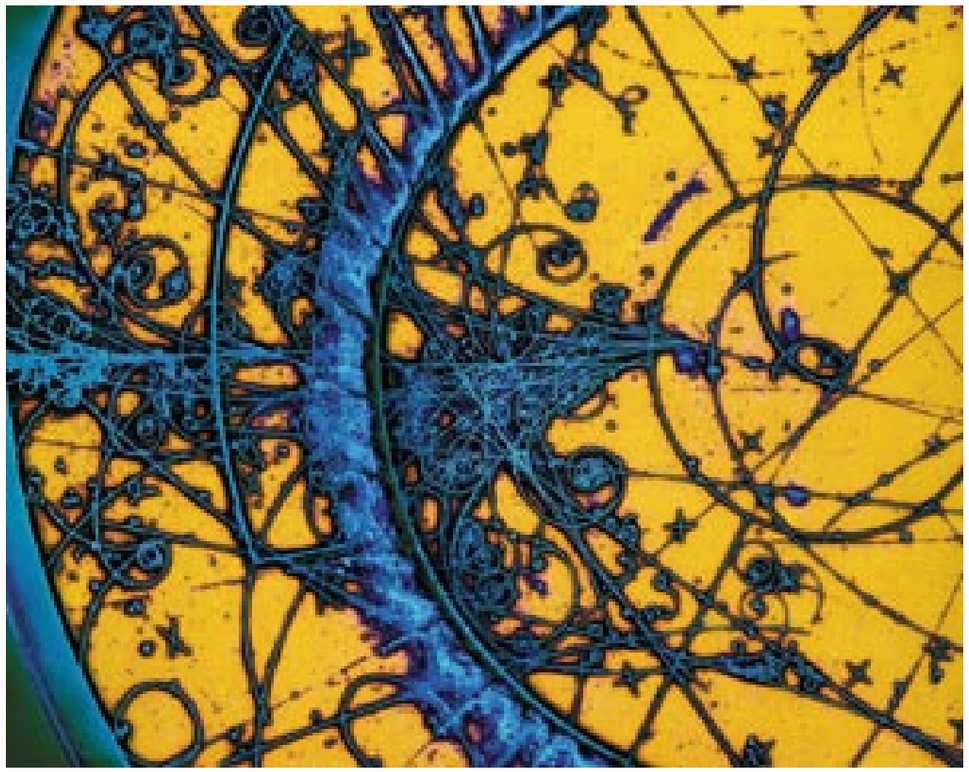
- create a more satisfactory image,
- or extract hidden information.

Image processing seeks algorithms that can carry out common image tasks automatically.

OVERVIEW: Sample Images







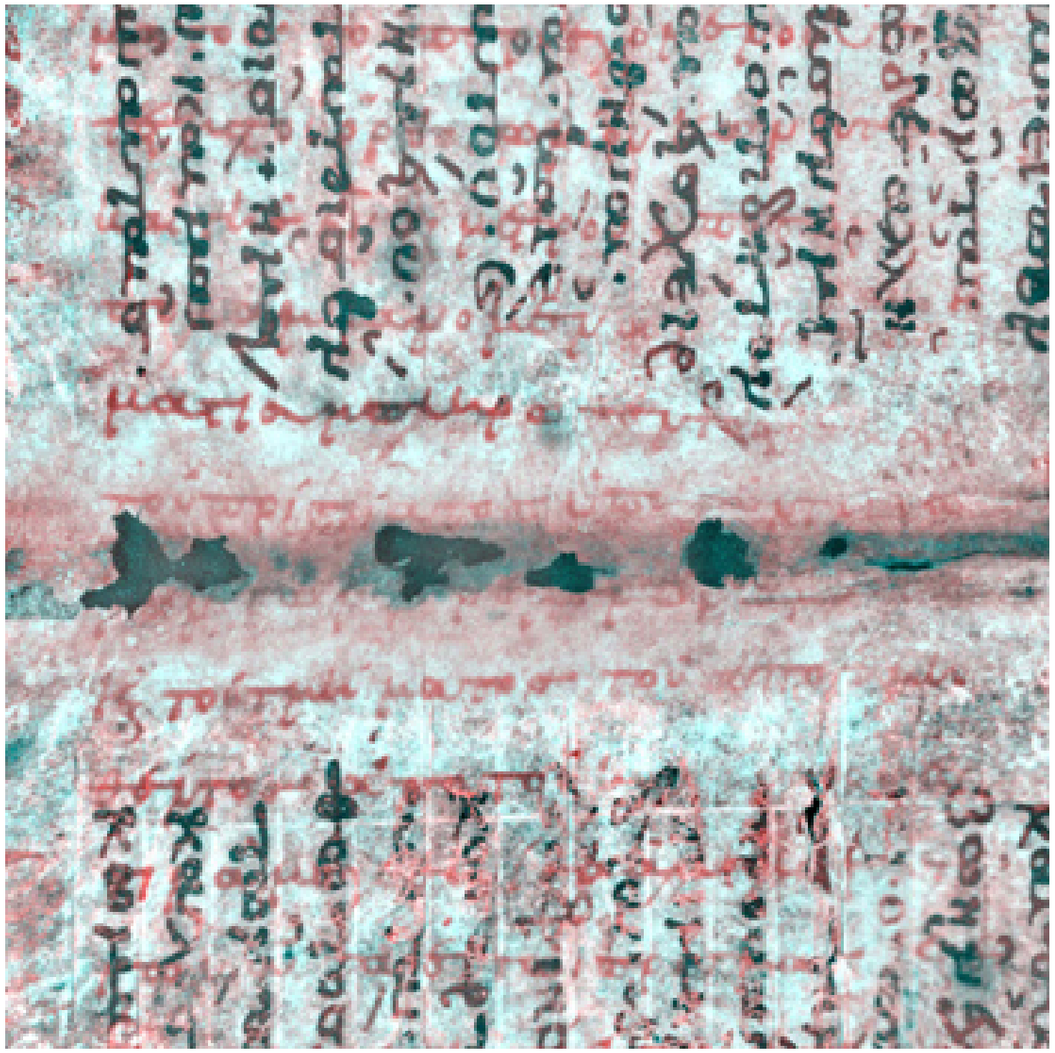
OVERVIEW: Sample Images

These images suggest the variety of objects for which some kind of image enhancement or analysis is desired:

1. The Mona Lisa; the image is too dark to see some details;
2. One of 24 slices of an MRI brain scan;
3. A satellite reconnaissance photo;
4. A color image damaged by noise;
5. An image of particle collisions in a bubble chamber.

The particle chamber, in particular, can generate millions of images, which must somehow be analyzed automatically.

OVERVIEW: Sample Images



OVERVIEW: Sample Images

The previous pair of images show a prayerbook under normal illumination, and the same prayerbook after images were made at multiple wavelengths, including X-rays, and combined.

The processed image reveals that under the text of the prayerbook are the traces of a lost manuscript of Archimedes.

For more information, go to "The Archimedes Palimpsest Project" at http://www.archimedespalimpsest.org

OVERVIEW: A View Becomes an Array

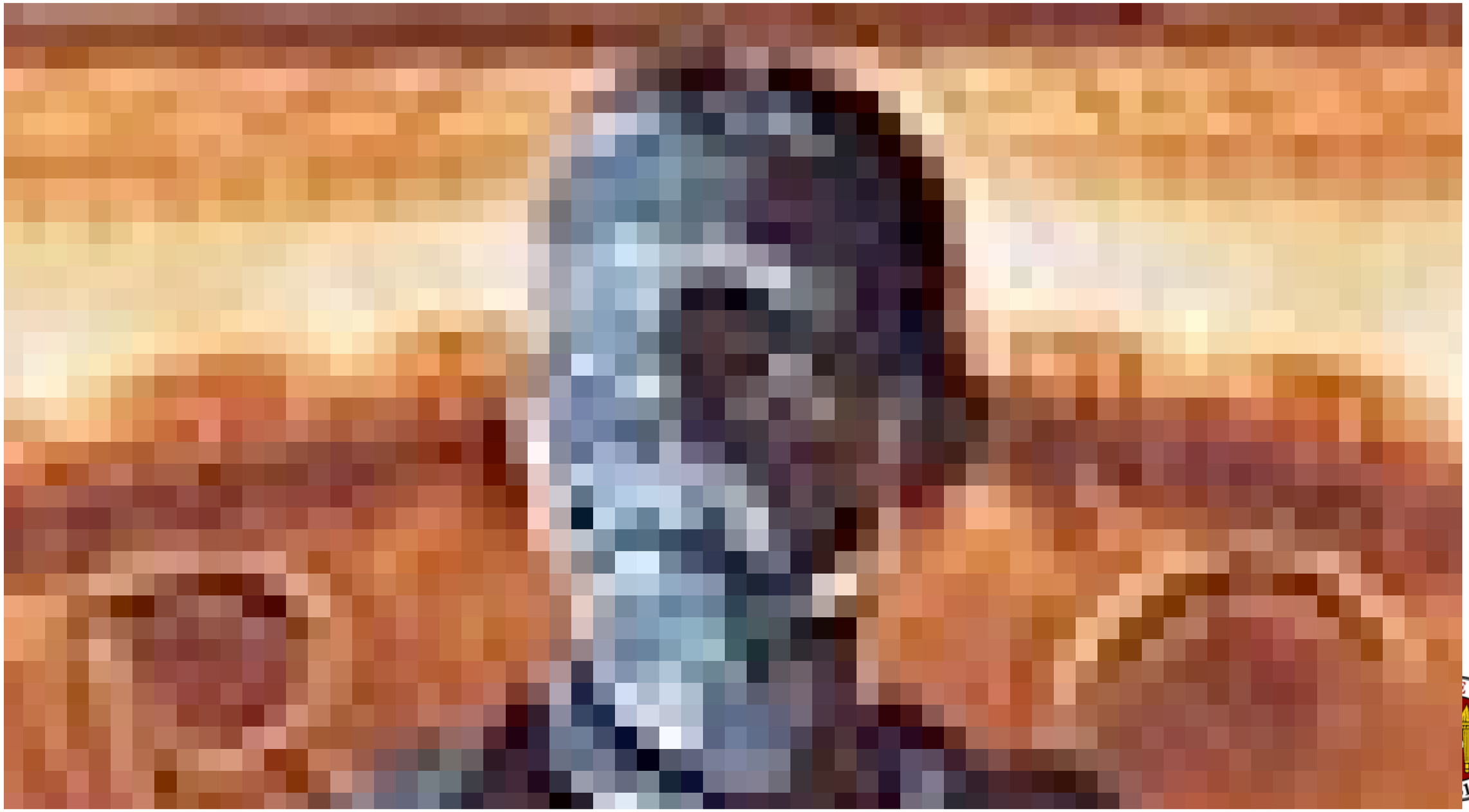
A 3D scene becomes a 2D perspective view.
The view is limited to a 2D rectangle.
The rectangle is discretized into “pixels”.
Each pixel is a single color, represented by numbers.

OVERVIEW: Pixels Must Disappear For Realism

Our picture elements are squares of color.
The eye will be very disturbed unless the pixels are small enough.
A coarse picture will do badly in representing lines and edges.
500 pixels in each direction are enough for small images.
This image is 541x200 pixels.

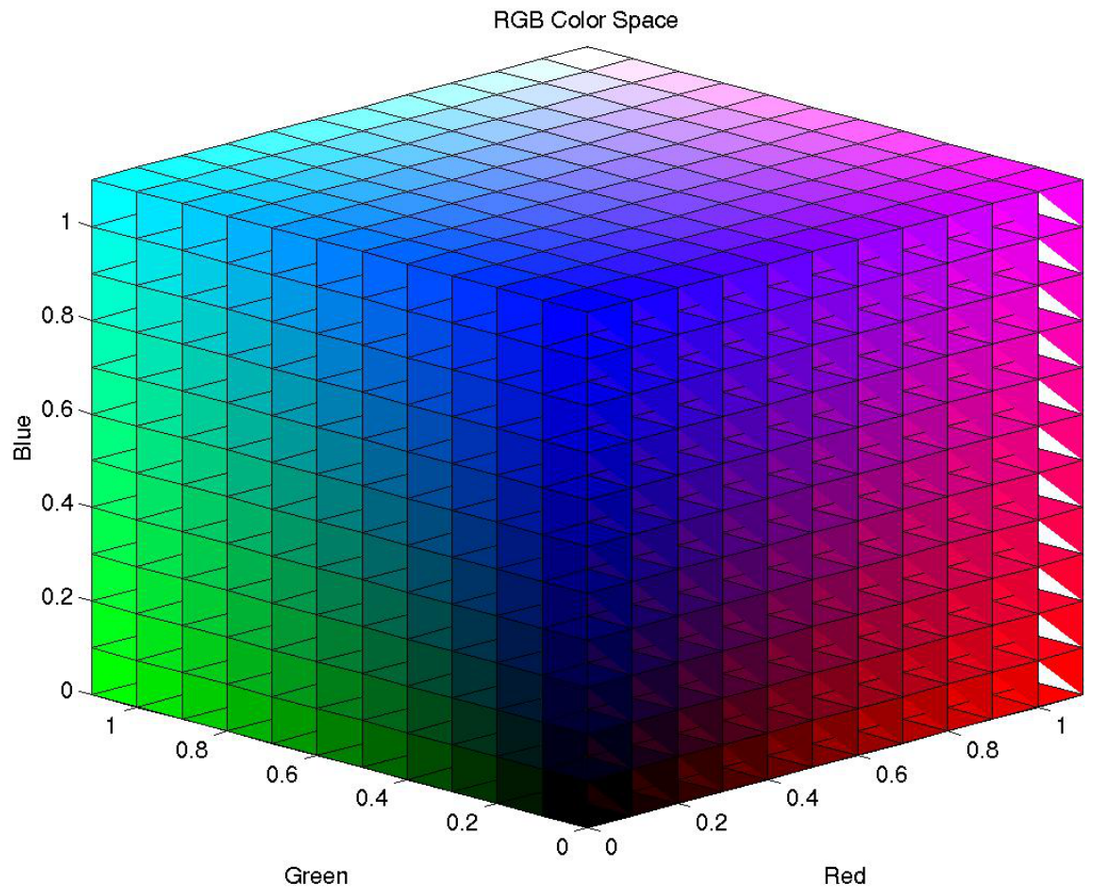
OVERVIEW: Pixels Must Disappear For Realism

This is really the same picture, but now your eye is very unhappy!



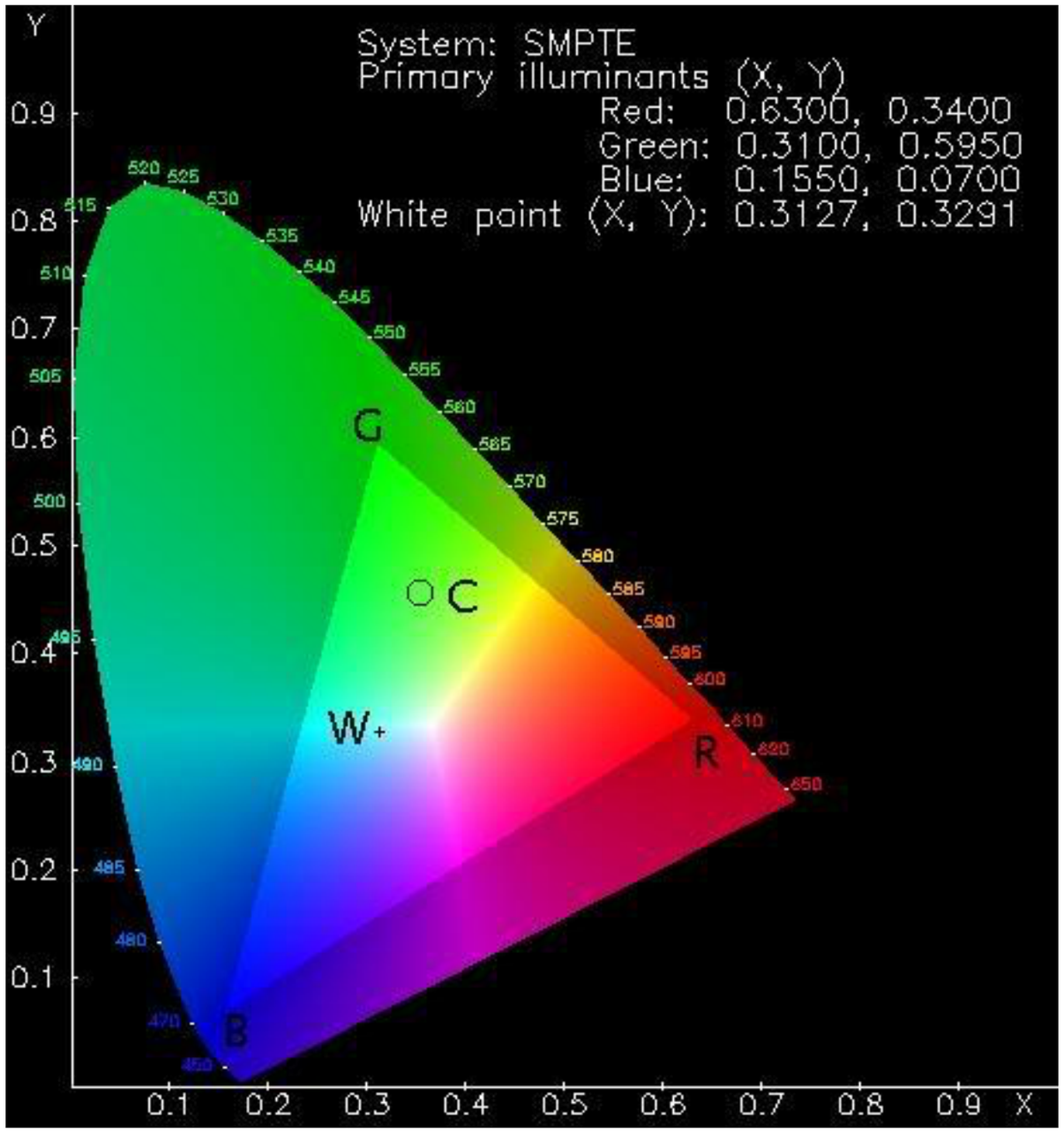
OVERVIEW: A Color Becomes an RGB Setting

Color is represented by red, green and blue settings.
Values might be real in [0,1], or integers between 0 and 255.
Matching real colors with computer display colors is hard.



OVERVIEW: A Color Becomes an RGB Setting

Some colors can't be represented by the RGB system at all.



OVERVIEW: Satellite Images

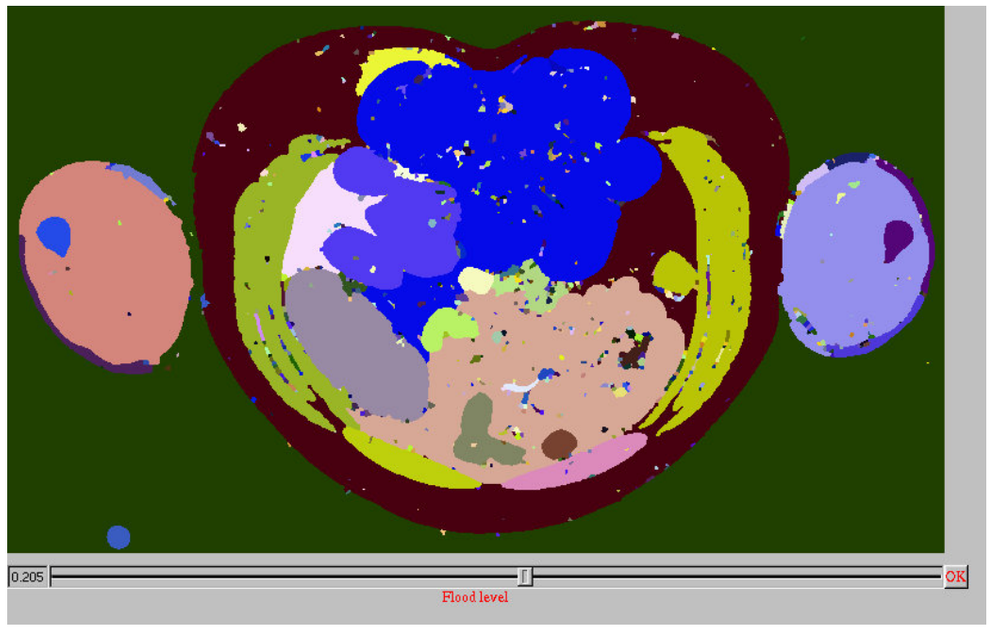
Satellites return images in visible, infrared or other wavelengths.

- Can you detect "line" structures such as roads?
- Can you detect "area" structures, such as buildings?
- Can you detect an airplane, in any orientation?
- Can you detect changes since the last picture was taken?

OVERVIEW: Medical Images

In medicine, an image can be sonogram, a single X-ray, a series of MRI slices forming a 3D image, or an animation from an fMRI.

- Can you detect irregular tissue by pixel values?
- Can you distinguish one big tumor from many small ones?
- Can you determine the seriousness of an infection by "integrating" the brightness of pixels, where white means very infected?
- Can you outline specific organs and track them as they move?



OVERVIEW: Facial Recognition

How can you computerize the recognition of a face? We do it so automatically that we can't even begin to explain the process. But it's easy to spot your friend in a picture you've never seen before, no matter what the expression.

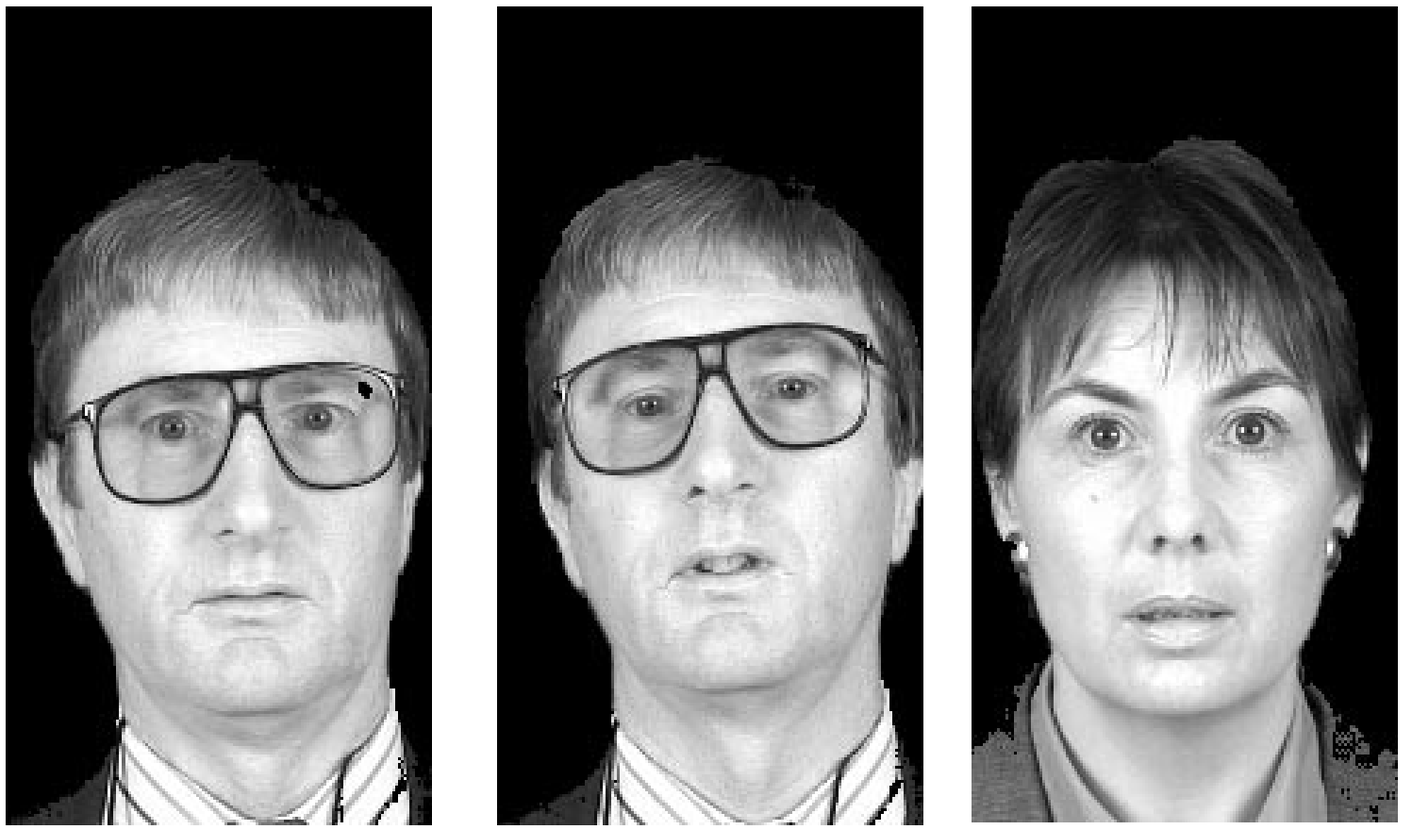


Image Algorithms

- Overview
- **Representing an Image**
- The Portable Gray Map Format
- Image Histograms
- Contrast Stretching
- Contrast Enhancement
- Salt and Pepper Noise
- Gaussian Noise
- Edge Detection
- Components
- Projects

REPRESENT: Representing an Image

In computational science, we are used to the idea of *discretization*; that is, the technique of representing a complicated "continuous" object by an orderly array of "discrete" numbers.

Let us begin with the following definition of an image, which we will have to expand later:

An image is an **M** by **N** array of integers between 0 and **MAXINT**.

Of course, we assume this array has some pictorial meaning, but for now it's just a bunch of numbers.

For now, we will also assume that the image is 'black and white" or actually, shades of gray. Once we understand how grayscale images work, color will not be hard to add in.

REPRESENT: The FEEP Example

Suppose our image was an array of **M** = 7 rows and **N** = 24 columns between 0 and **MAXINT** = 15:

```
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
0  3  3  3  3  0  0  7  7  7  7  0  0 11 11 11 11  0  0 15 15 15 15  0
0  3  0  0  0  0  0  7  0  0  0  0  0 11  0  0  0  0  0 15  0  0 15  0
0  3  3  3  0  0  0  7  7  7  0  0  0 11 11 11  0  0  0 15 15 15 15  0
0  3  0  0  0  0  0  7  0  0  0  0  0 11  0  0  0  0  0 15  0  0  0  0
0  3  0  0  0  0  0  7  7  7  7  0  0 11 11 11 11  0  0 15  0  0  0  0
0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0
```

If this array represents an image, what should we expect to see if we could display it?

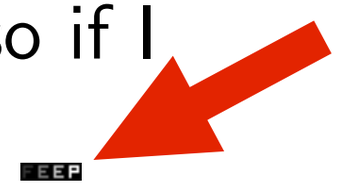
REPRESENT: The FEEP Example Displayed

The relationship between the array and the picture starts to make some sense!

Rows and columns of numbers <==> rows and columns of pixels.
0 means black, 15 means white.

REPRESENT: The FEEP Example Explained

Actually, the FEEP example is just 7 pixels by 24 pixels, so if I had not zoomed in, you would not have seen it!



Since we used the 16 integers 0 to 15, a range of 2^4 values, each pixel could be described by 4 bits of information, so this is called a *4-bit grayscale image.*

A 1-bit image corresponds to a true black and white picture.

For grayscale images, a more common choice uses *8 bit grayscale*, so that the values range from 0 to 255.

The available range of grays or colors is known as the color depth of an image. Things will become more complicated when we look at actual color images!

REPRESENT: The FEEP Example Explained

For our work with gray scale images, we can assume our gray scale ranges from 0 to 255.

For an image to properly show up on our computer screen without zooming, we will typically expect somwhere between 300 to 1200 pixels horizontally and vertically. You can see that a typical image might contain a million data values.

QUIZ: What happens to our FEEP image if we leave the data alone but change the maximum allowed gray value from 15 to 31?

QUIZ: Suppose we wanted to make a "negative" of the FEEP image, so that dark and light are swapped. How can we do this?

REPRESENT: Integers Become Real Numbers

In the FEEP example, 0 was black and 15 was white.

In an 8 bit grayscale, 0 is still black but white is now 255.

While it is easier to store gray values as integers, it makes more sense to think of them as real numbers between 0 and 1. That way black is always 0.0 and white is always 1.0.

To determine the shade **S** indicated by a gray scale value **G**, write

```
S = G / MAXINT;
```

or, in C, where integer division only gives integer results:

```
S = ( float ) G / ( float ) MAXINT;
```

REPRESENT: Where Does Gray Come From?

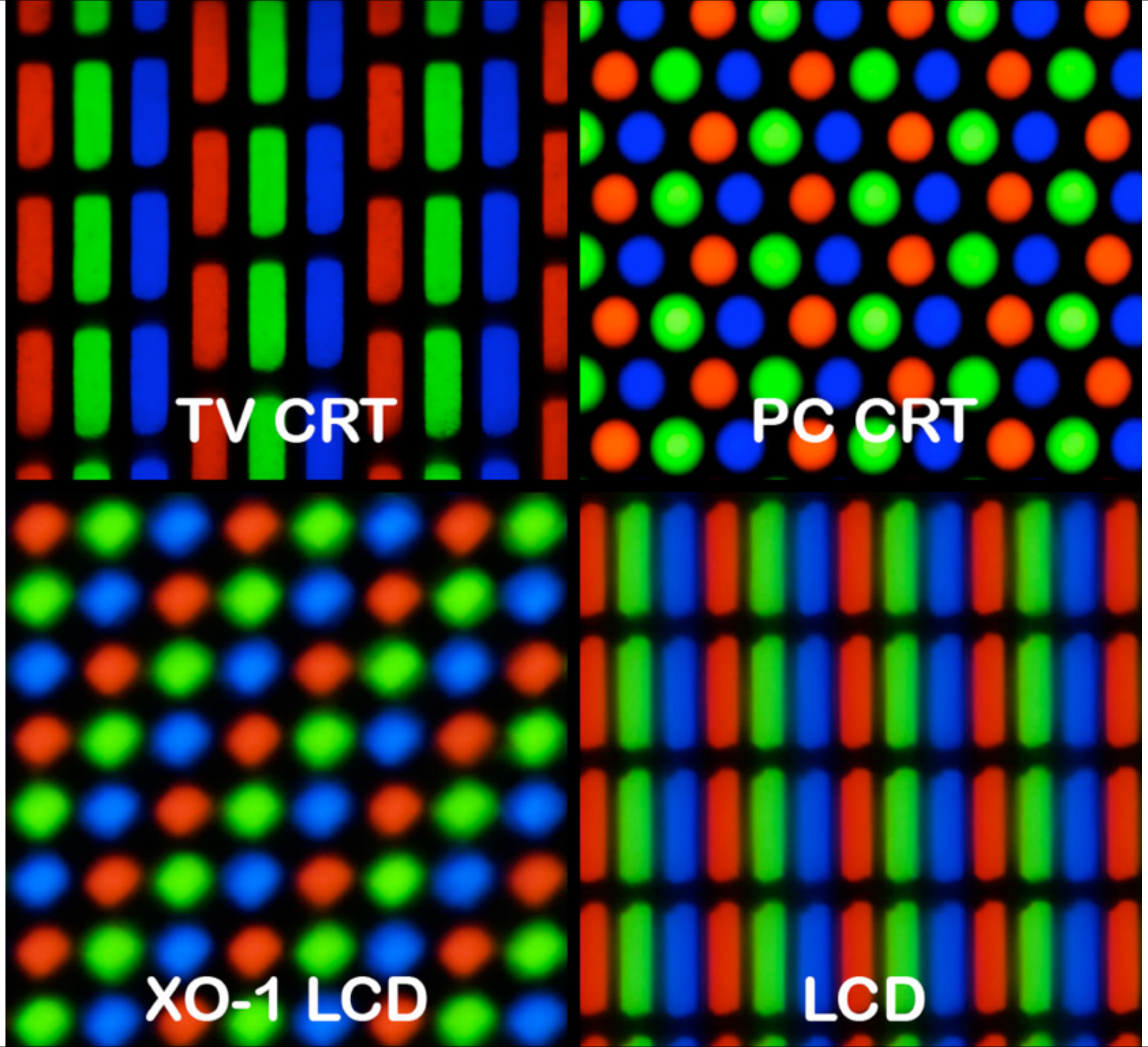
Our FEEP example image involves black, white, and grays.

But our computer pumps out red, green, and blue.
Where did the gray come from, and where did the color go?
R=G=B=0, means no light, so black...that's easy to understand.

The relationship between the R, G and B values and the corresponding light intensity have been carefully calibrated.

Surprisingly, if R, G and B are all at full intensity (a real value of 1.0, or perhaps an integer value of 255) the result is white light.

More interesting, as long as R=G=B, your eyes will register the displayed color as **gray**.


TV CRT
PC CRT
XO-1 LCD
LCD

REPRESENT: Where Does Gray Come From?

So, if we know we are doing a grayscale image, we could store each pixel as a single number...as long as we understand that that value is to be used to set R, G and B.

If we must use a color format for a grayscale image, we have to specify the same value three times, once for each color channel.

Many image file formats, recognizing that a gray image really only needs 1/3 the data of a full color image, specifically include an option to store the image as grayscale.

This is one example of image compression; if you think about the fact that a single photo image can contain 100 MegaBytes of pixels, (assuming 1 pixel = 1 Byte) that a movie requires about 30 frames per second, and lasts 90 minutes, we have:

$$90 \text{ min} \approx 5000 \text{ sec} \approx 150,000 \text{ frames} \approx 15 \text{ TeraBytes}$$

so now you know one huge area of image compression!

REPRESENT: Unsigned Short Integers

Since images require a lot of data, it makes sense to cut down on the size of each data item, if possible. In most computer languages, an "integer" or "int" corresponds to 32 bits; on MATLAB, an integer is stored in 64 bits.

We only need 8 bits, so we could use the C datatype called a **short**; in MATLAB, the corresponding item is an **int8**.

But integers in images are always positive, while in computers we usually have a signed representation for integers that "uses up" one bit for the sign, so that 8 bits represent the range -128 to +127.

Luckily, C allows you to declare something called an unsigned short and MATLAB has a datatype called uint8, both of which will create integers that only use 8 bits and range from 0 to 255.

We will need to know this when we "pack" and "unpack" images.

Image Algorithms

- Overview
- Representing an Image
- **The Portable Gray Map Format**
- Image Histograms
- Contrast Stretching
- Contrast Enhancement
- Salt and Pepper Noise
- Gaussian Noise
- Edge Detection
- Components
- Projects

PGM: The Difference Between Images and Formats

It's probably best if we think of an image as an abstract, mathematical object, something like the following:

- **An image** is an M by N array of pixel values.
- **A grayscale image** pixel value determines a shade of gray.
- **An RGB color image** pixel value determines R, G and B levels that specify an RGB color.

We'll assume that once an image is inside the computer, the computer knows what it's dealing with and how to display it.

All the details about storage, row and column ordering, maximum entries, real or integer arithmetic, order of R, G, and B, and compression are left to decide by the **file format** we use.

PGM: Images Must be Formatted

An image file is a file that represents the information in an image.

To "understand" the file, we need to know the file format, which allows us to reconstruct the image.

Why do we have to make things so complicated? Why can't we simply agree to store an image by rows, then columns, in the order (R,G,B), and with pixel values between 0 and 255?

It takes a moment to realize that images are used across an incredible array of human activity. The needs of a photographer, an animator, a book publisher, a mathematician, a medical technician are vastly different.

The primary problem is that the raw information in a typical image is both enormous and redundant.

PGM: Different Formats

- Compression requires a sophisticated storage scheme;
- Some images include huge areas of the same color;
- Images vary in their color resolution. For true black and white images, color is 0 or 1. For high-quality images, 255 color shades ("8 bit depth") is not enough.
- Some images are mathematical or technical; they are best represented by lines and solid polygons of color. Their storage can be vastly reduced by a description of how to draw the figure, rather than by pixel values;
- Some images are actually predominantly text files. Because text is examined and printed at many magnifications, a single character is not well described by its pixel representation at a single resolution.

PGM: Run Length Encoding

As a simple example of how an image can be compressed, consider the technique called run length encoding or **RLE**. Given a vector of pixel data, we create a new array of pairs of data. The first item is a pixel value, and the second item is the number of times to repeat this value.

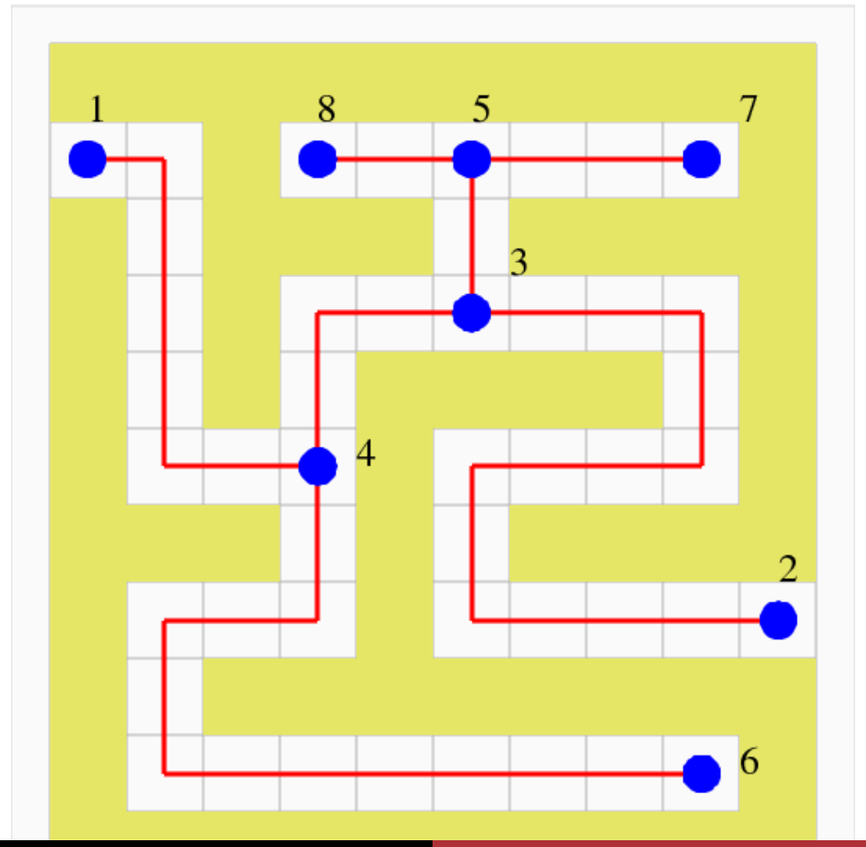
The first three lines of our FEEP file use 72 values.
The RLE version needs 42.

```
0  0  0  0  0  0  0  0  0  0  0  0  0   0   0   0   0  0  0   0   0   0   0  0
0  3  3  3  3  0  0  7  7  7  7  0  0  11  11  11  11  0  0  15  15  15  15  0
0  3  0  0  0  0  0  7  0  0  0  0  0  11   0   0   0  0  0  15   0   0  15  0


0  24
0  1  3  4  0  2  7  4  0  2  11  4  0  2  15  4  0  1
0  1  3  1  0  5  7  1  0  4  11  1  0  5  15  1  0  2  15  1  0  1
```

PGM: Vector Graphics

PostScript is an image file format that describes text or a line drawings economically.

```
1.0 0.0 0.0 line_rgb
0.5 9.5 moveto
1.5 9.5 drawto
1.5 5.5 drawto
3.5 5.5 drawto
```




```
%!PS-Adobe-2.0 EPSF-1.2
%%Creator:Adobe Illustrator(TM) 1.0b2-
%%Title:golfer art+
%%CreationDate:1/6/87 9:32 AM
%%DocumentFonts:Helvetica-Bold
%%BoundingBox:7 31 577 726
%%TemplateBox:0 -48 576 672
%%EndComments
/golfersave save def            % prevent left
100 dict begin
/q{bind def}bind def
/Q{load def}q
/x{exch def}q
/X/def Q
/g{/_g x/p{_g setgray}X}q
/G{/_G x/P{_G setgray}X}q
/k{/_b x/_g x/_r x/p{_r _g _b setrgbcolor}X}q
/K{/_B x/_G x/_R x/P{_R _G _B setrgbcolor}X}q
/d/setdash Q
/i/setflat Q
/j/setlinejoin Q
/J/setlinecap Q
/M/setmiterlimit Q
/w/setlinewidth Q
/_C{.25 sub round .25 add}q
/_c{transform _C exch _C exch itransform}q
/c{_c curveto}q
/C/c Q
/v{currentpoint 6 2 roll _c curveto}q
```

PGM: Living with Multiple Formats

You will often have image information which you need to examine, or to convert to another format.

- Everyone thinks of **Adobe PhotoShop** for this task;
- The Gnu **GIMP** program is similar to PhotoShop, and allows you to view, edit, and convert images.
- The ImageMagick program **convert** can be used to convert images from one format to another.
- MATLAB's **Image Processing Toolbox** reads images, allows you to edit them, includes many tools for common tasks, and can convert between various formats.

PGM: Common Image Formats

Image file formats you may encounter include:

- **BMP**, Microsoft Bit Map;
- **GIF**, once popular for web graphics;
- **JPEG**, what comes out of your digital camera;
- **PBM**, PGM, PPM, portable B/W, gray, and color;
- **PDF**, for documents to be published;
- **PNG**, an open-source replacement for GIF;
- **PostScript**, for documents to be published;
- **TIFF**, a high-quality format.

PGM: The Portable Gray Map

One of the simplest formats for grayscale images is called the *Portable Gray Map* or PGM format.

It is a perfect beginner's format, since it corresponds very closely to our logical representation of an image, it is specifically designed for grayscale images, and it comes in both an ASCII version (which is easy to print or edit) and a binary version (which saves space).

When we are ready for color, there is a related PPM format.

For more information, see the NETPBM webpage at http://netpbm.sourceforge.net/

PGM: The FEEP Example

Our FEEP example could be stored as the following PGM file:

```
P2
# This is an ASCII PGM file.
24 7
15
0 0 0 0 0 0 0 0 0 0 0 0  0  0  0  0 0 0  0  0  0  0 0
0 3 3 3 3 0 0 7 7 7 7 0 0 11 11 11 11 0 0 15 15 15 15 0
0 3 0 0 0 0 0 7 0 0 0 0 0 11  0  0  0 0 0 15  0  0 15 0
0 3 3 3 0 0 0 7 7 7 0 0 0 11 11 11  0 0 0 15 15 15 15 0
0 3 0 0 0 0 0 7 0 0 0 0 0 11  0  0  0 0 0 15  0  0  0 0
0 3 0 0 0 0 0 7 7 7 7 0 0 11 11 11 11 0 0 15  0  0  0 0
0 0 0 0 0 0 0 0 0 0 0 0 0  0  0  0  0 0 0  0  0  0  0 0
```

No line should be longer than 70 characters.
No data value should be less than 0 or greater than the declared maximum (which is 15 for this file).

PGM: The FEEP Example

To get a copy of the ASCII PGM version of the FEEP image:

`//people.sc.fsu.edu/~jburkardt/data/pgma/feep.ascii.pgm`

or, for the binary version:

`//people.sc.fsu.edu/~jburkardt/data/pgmb/feep.pgm`

PGM: Displaying an Image

MATLAB's Image Processing Toolbox includes the imshow command which can display an image file for us:

imshow (*'feep.pgm'*) <-- file in current directory.

imshow (*'../../data/pgmb/feep.pgm'*) <-- elsewhere.

You can blowup a tiny image using the magnifying glass on the toolbar, or specify a magnification percent on the command line:

imshow (*'feep.pgm'*, **'InitialMagnification'**, *400*)

MATLAB rescales data to [0,255]. We can limit the display range:

imshow (*'feep.pgm'*, **'DisplayRange'**, *[100, 150]*)

PGM: Reading an Image

MATLAB's Image Processing Toolbox includes the imread command which can read an image file, creating an array we can manipulate. An M by N gray scale image becomes an M by N MATLAB array.

```
u = imread ( 'feep.pgm' );    <-- Use semicolon!
```

Note that **u** is an array of unsigned 8-bit integers. To make a numeric copy that we can work with, try

```
du = double ( u );
```

To convert the other back, try

```
udu = uint8 ( du );
```

PGM: Unsigned 8 Bits

You will make many mistakes thinking that unsigned 8 bit integers are ordinary MATLAB numbers.

The command **imshow (u)** assumes that **u** is an image array, not a filename, (because it's not quoted).

```
u = imread ( 'feep.pgm' );
u                       <-- Print the image data.
imshow ( u )            <-- Show the image data.
2 * u                   <-- Oops, no number is bigger than 255!
u + 1000                <-- All numbers become 255.
1 - u                   <-- All numbers become 0 or 1
255 - u                 <-- Finally, something useful!
                            Do an imshow ( 255 - u )!
```

A uint8 will refuse to move outside the range [0,255], and only a conversion function like **double()** can change that!

PGM: Writing a File

MATLAB's Image Processing Toolbox includes the imwrite command which can write an M by N MATLAB array of uint8 values as an image file. If we use the suffix **.pgm**, the file is created as a PGM file. It will be a *binary* PGM file unless we include the "ASCII" option.

If we assume **u** is a MATLAB uint8 array, then:

```
imwrite ( u, 'feep1.pgm' );  <-- creates binary PGM.

imwrite ( u, 'feep2.pgm', 'Encoding', 'ASCII' );
  <-- creates an ASCII file.
```

Using the same data array **u**, try both these commands and compare the size of the output files.

Image Algorithms

- Overview
- Representing an Image
- The Portable Gray Map Format
- **Image Histograms**
- Contrast Stretching
- Contrast Enhancement
- Salt and Pepper Noise
- Gaussian Noise
- Edge Detection
- Components
- Projects

HISTO: Histogramming an Array

Our first attempts to understand an array of data require getting some statistics, such as the maximum and minimum values, or a visual display such as a histogram, that shows us simply the frequency of occurrence of each value.

Let's see what we can find out about a simple snapshot, which is given to us as a binary PGM file *snap.pgm*.

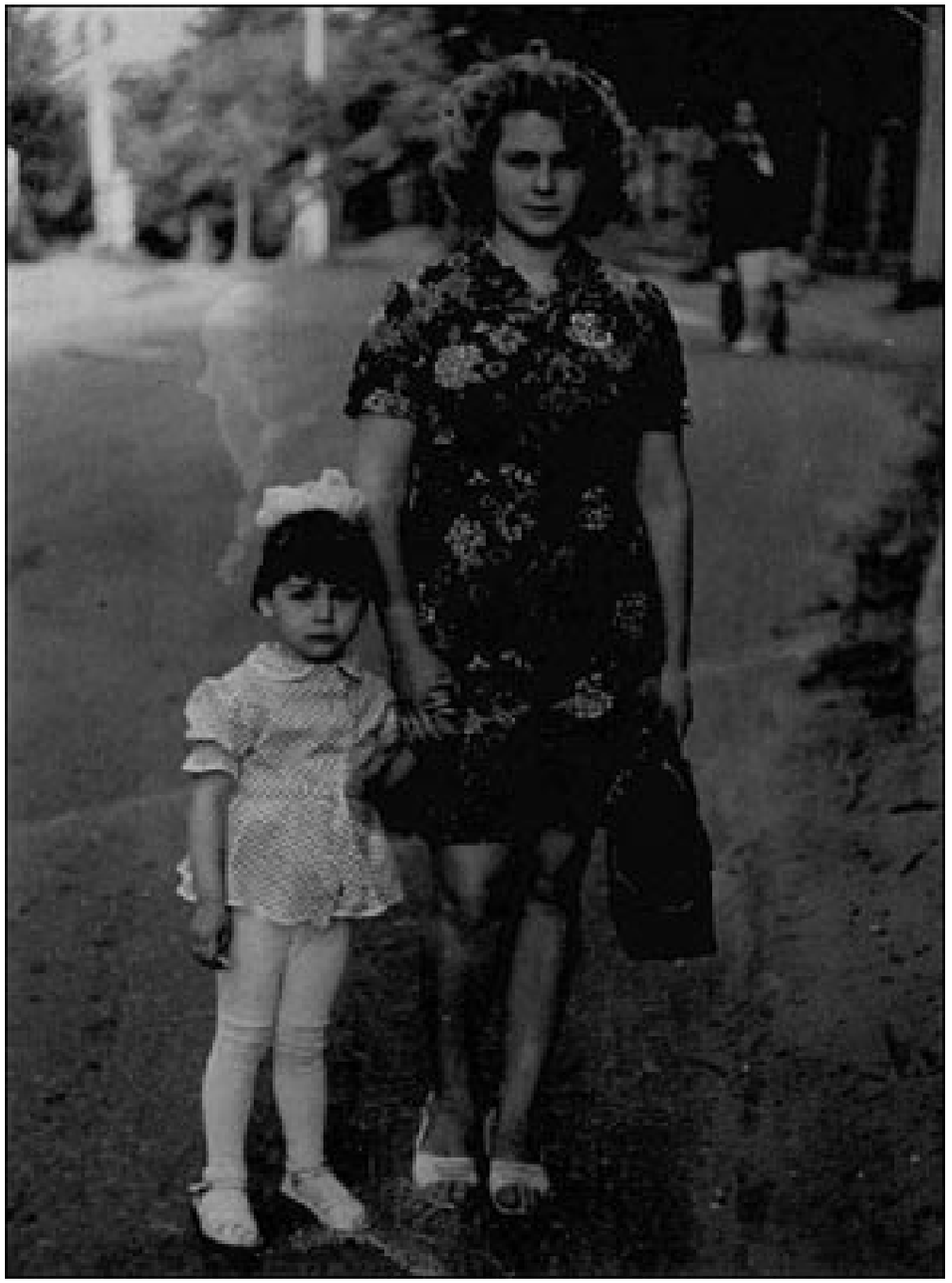
Let's begin by reading the file in, and displaying it:

```
g1 = imread ( 'snap.pgm' );
imshow ( g1 );
```

http://people.sc.fsu.edu/~jburkardt/data/pgmb/snap.pgm

HISTO: The SNAP Example

This certainly seems a dark image!



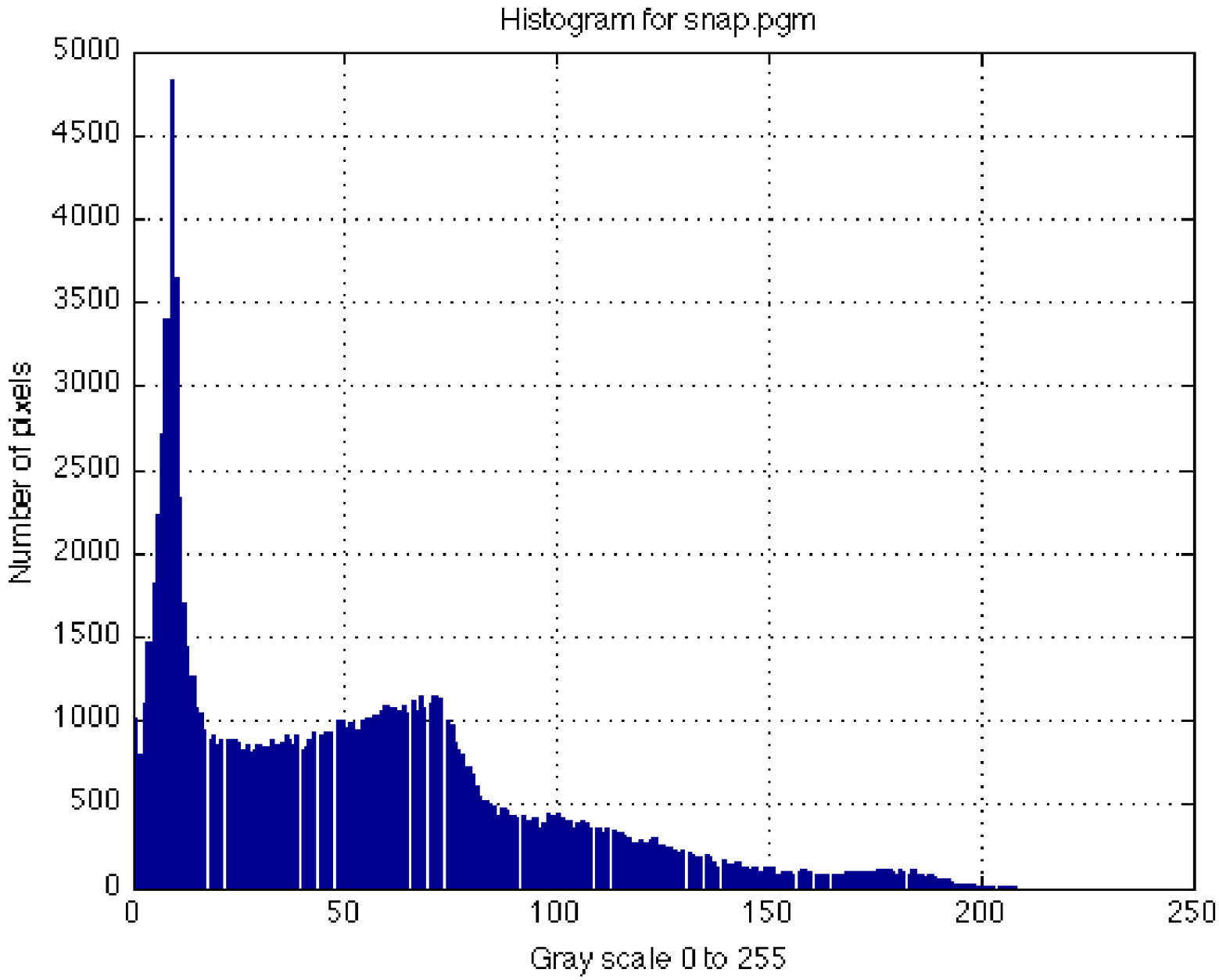
HISTO: The HIST command

MATLAB has a **hist** command which can display a histogram of data. However, it only works on a vector, and the vector has to contain data of type *double*.

```
g2 = double ( g1 );   <-- convert to double;
size ( g2 )           <-- returns [ 400, 295 ];
g3 = reshape ( g1, 400 * 295, 1 );  <-- g3 a vector;
hist ( g3, 256 );     <-- use 256 bins.
```

HISTO: The SNAP Histogram

This shows how dark the picture is...or how the picture is dark.



HISTO: Experiment 1: Lighten Up!

Let's try to make the picture lighter. The simplest way might be to double all the grays between 0 and 127, and to set all higher grays to 255.

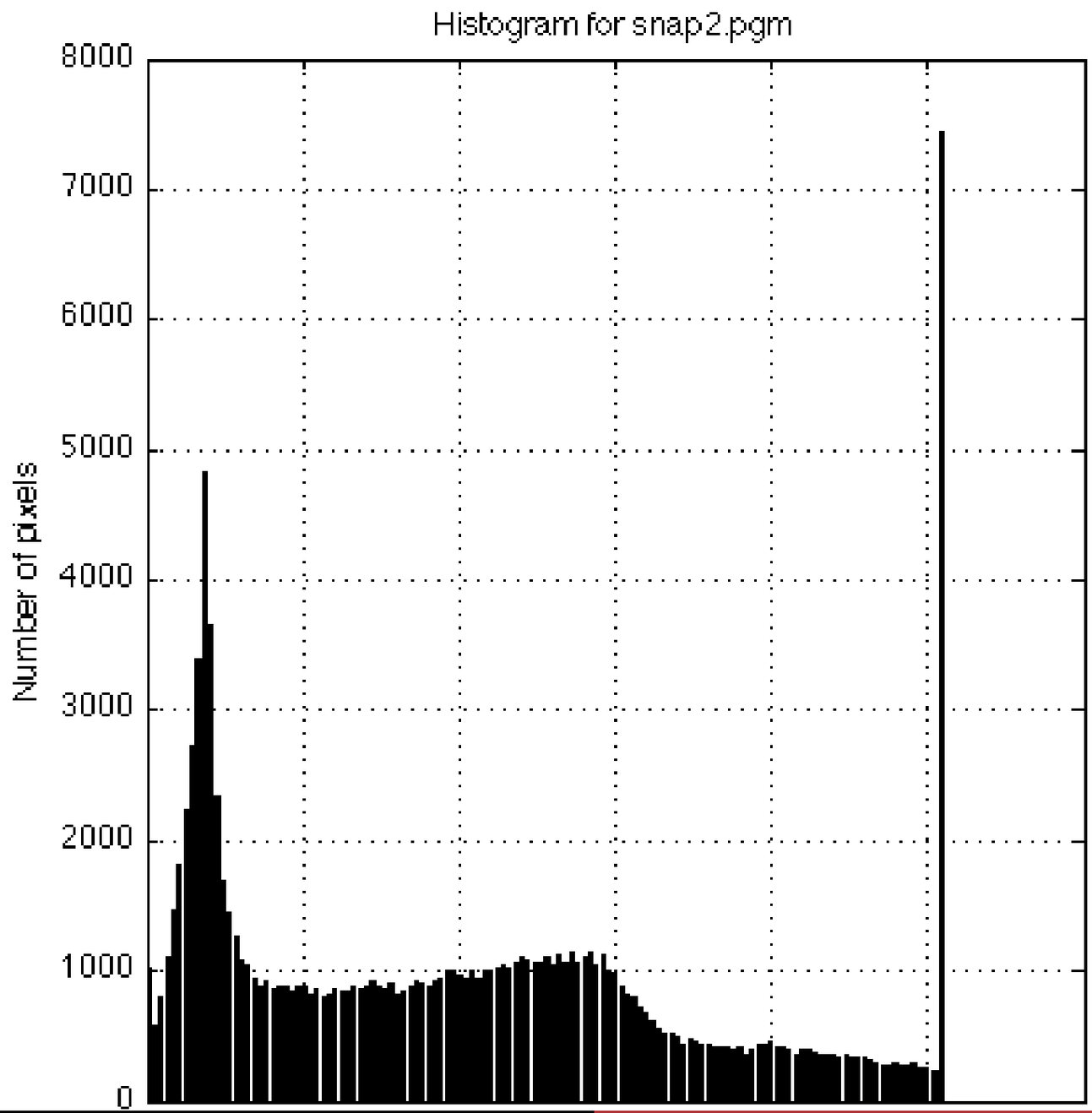
```
g4 = g2;                    <-- copy.
i1 = g4 < 128;              <-- dark grays.
i2 = 128 <= g4;             <-- light grays.
g4(i1) = 2 * g4(i1);        <-- double darks.
g4(i2) = 255;               <-- set lights to white.
g5 = uint8 ( g4 );          <-- back to uint8.
imshow ( g5 );              <-- display.
```

HISTO: The SNAP2 Image

The image seems brighter!

HISTO: The SNAP2 Histogram

The information was mostly in the dark range. Stretching it out made it easier to see.



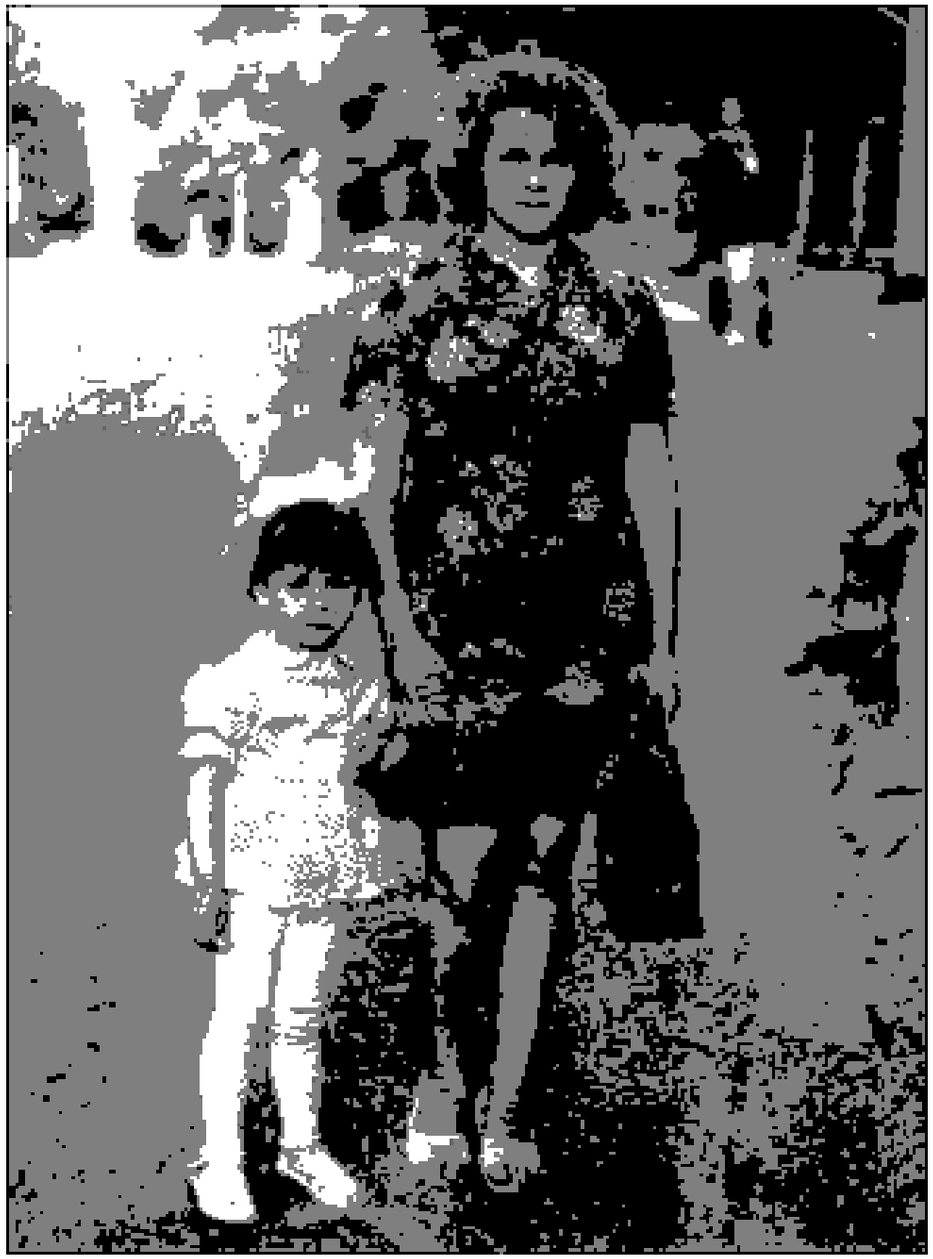
HISTO: Experiment 2: Three Colors!

Did our picture get better because we have more grays to look at, or because the darks and lights were more spread apart? One way to think about this question is to reduce the picture to using just three shades of gray. Based on the histogram, it seems as though we could break the range into 0:25:100:255.

```
g6 = g2;                              <-- copy.
i1 =                 g6 <  25; <-- dark grays.
i2 =  25 <= g6 & g6 < 100; <-- medium grays.
i3 = 100 <= g6;                       <-- light

g6(i1) = 0;                           <-- black.
g6(i2) = 127;                         <-- gray.
g6(i3) = 155;                         <-- white.

g7 = uint8 ( g6 );                    <-- back to uint8.
imshow ( g7 );                        <-- display.
```

HISTO: The SNAP3 Image

Black, white, and one gray are enough.



HISTO: What Happened to "Automatic"?

You should be a little concerned that the examples we have looked at have not been automatic. In both cases, I looked at the histogram, and then said, "Let us change the data as follows." I did not try to write out my reasoning as an algorithm that even a computer could understand.

Some sort of automatic procedure would be necessary if we are going to ask a computer to clean up our pictures.

You may already have seen one such example of automatic choice of a small number of colors for an image, because this was an example in the "clustering" section of this course.

We will use a technique called K-Means to try to automatically choose a small number of gray scales.

HISTO: The K-Means Algorithm

The K-Means algorithm is given **N** items of data, and seeks **K** "representative" values, in such a way that every data item can be assigned to a nearby representative.

If we replace each color or shade by its representative, then we have reduced the "color complexity" of the image from **N** to **K**. If this is done in a systematic way, less memory is required to store the image.

HISTO: The K-Means Algorithm

The algorithm initializes the "centers" **C** at random.

For each data point **P(I)**, we set **PTOC(I)** to be the nearest **C**.

The values of **C** are now replaced by the averages of the values of **P** assigned to them.

The process repeats until no **P** gets reassigned after the latest update.

We'll try this on a file stored at:
http://people.sc.fsu.edu/~jburkardt/data/pgmb/casablanca.pgm

HISTO: MATLAB Code

```
[ m, n ] = size ( g );
p = double ( g );                  <-- G converted to double.
p = reshape ( p, m * n, 1 );       <-- P becomes a vector;

k = 5;                             <-- Choose number of centers
[ c, ptoc ] = kmeans ( p, k );     <-- Centers, assignments.
ptoc = round ( ptoc );             <-- Want integer centers.
p = ptoc(c);                       <-- Set P to center.

p = reshape ( p, m, n );           <-- P goes back to an array.
g = uint8 ( p );                   <-- G is 8-bit.
```

HISTO: Casablanca, 256 colors

Original image

HISTO: Casablanca, KMEANS chooses K = 10 colors

Image using 10 shades

HISTO: Casablanca, KMEANS chooses K = 3 colors

HISTO: KMEANS Startup Problem

You might see this KMEANS error message:

```
g = imread ( 'image.png' );
p = double ( g );
[ m, n ] = size ( p );
p = reshape ( p, m * n, 1 );
[ c, ptoc ] = kmeans ( p, 10 )
??? Error using ==> kmeans>batchUpdate at 435
Empty cluster created at iteration 1.
Error in ==> kmeans at 336
        converged = batchUpdate();
```

This can be fixed by:

```
[ c, ptoc ] = kmeans ( p, 10, 'Start', 'uniform' );
```

Try "*help kmeans*" for details!

HISTO: Summary

The gray level histogram gives a useful summary of where the "action" is in the picture.

Contrast stretching is the technique of spreading out the gray scale over the range containing most of the information.

Quantization or color depth reduction is the process of reducing the number of colors or shades used in an image. The reduced set can be chosen by hand, or by a clustering technique such as K-Means. This is one way to compress an image.

QUIZ: How many bits would we need to describe the "colors" of the SNAP3 image? If we store the image as a sequence of these color descriptors, what important thing must we also supply? *(Hint: the second word is "map"!)*

Image Algorithms

- Overview
- Representing an Image
- The Portable Gray Map Format
- Image Histograms
- Contrast Stretching
- **Contrast Enhancement**
- Salt and Pepper Noise
- Gaussian Noise
- Edge Detection
- Components
- Projects

ENHANCE: Local Processing

In contrast stretching, we completely ignored the "geometry" of the picture. We counted the pixels of each shade, but we didn't care what pixels were next to others. We would have treated a checkerboard the same as a picture that was black on one half and white on the other.

In some images, the contrast problem occurs because an interesting feature is only indicated by a small shade change as we move from one pixel to another. This problem might be local, that is, confined to a small region of the image. Contrast stretching would be unlikely to fix it.

Let's look at the following picture:
http://people.sc.fsu.edu/~jburkardt/data/pgmb/surf.pgm

ENHANCE: The SURF Image

Some things in this picture seem indistinct or washed out.

ENHANCE: Local Differences

If a patch of the image seems to have a feature which we can't quite make out, then the pixels there must differ in shade. We can't see what's going on so well because the pixels don't differ enough. We might be able to exaggerate the difference in order to bring out the almost hidden features.

Imagine, for example, a globe that includes bumps and grooves that represent the hills and valleys of Earth. A globe that is true to scale would be extremely smooth, and we couldn't see or feel Mt Everest.

In order to actually see features, we would want to exaggerate their height or depth, to make up for the limitations of our senses.

ENHANCE: A Raised Relief Globe

ENHANCE: Estimating the Local Baseline

To exaggerate the "height" of a pixel, we have to have a numerical measurement of it. A standard way of doing this is to compare the pixel's shade to the average of the neighbors. A very simple neighborhood would be the pixels immediately left, right, above and below.

For a pixel with index (I,J), this would suggest:

```
                                  P(I-1,J)
Average = 1/4 * ( P(I,J-1)                  P(I,J+1) )
                                  P(I+1,J)

Height = P(I,J) - Average
```

For smoother results, we might average even more neighbors.

ENHANCE: Evaluating and Increasing the Height

If the pixel P(I,J) was going to "fit in" with its neighbors, we might expect it to equal the average value. But if something interesting is happening, it might be higher (or lower) than that value.

We can assign a numeric value to this "height":

```
Height(I,J) = P(I,J) - Average
```

Therefore, we can think of the pixel's value as the local average plus its height:

```
P(I,J) = Average(I,J) + Height(I,J)
```

and the nonzero height is the reason this pixel is interesting.

ENHANCE: Evaluating and Increasing the Height

If "height" makes a pixel more interesting, than we might think we can make the pixel "twice as interesting" by doubling its height, to exaggerate its difference from the local average.

Let's make the amount of exaggeration a parameter **S** so we remember we can set it to anything. Our urge to double the height would be expressed this way:

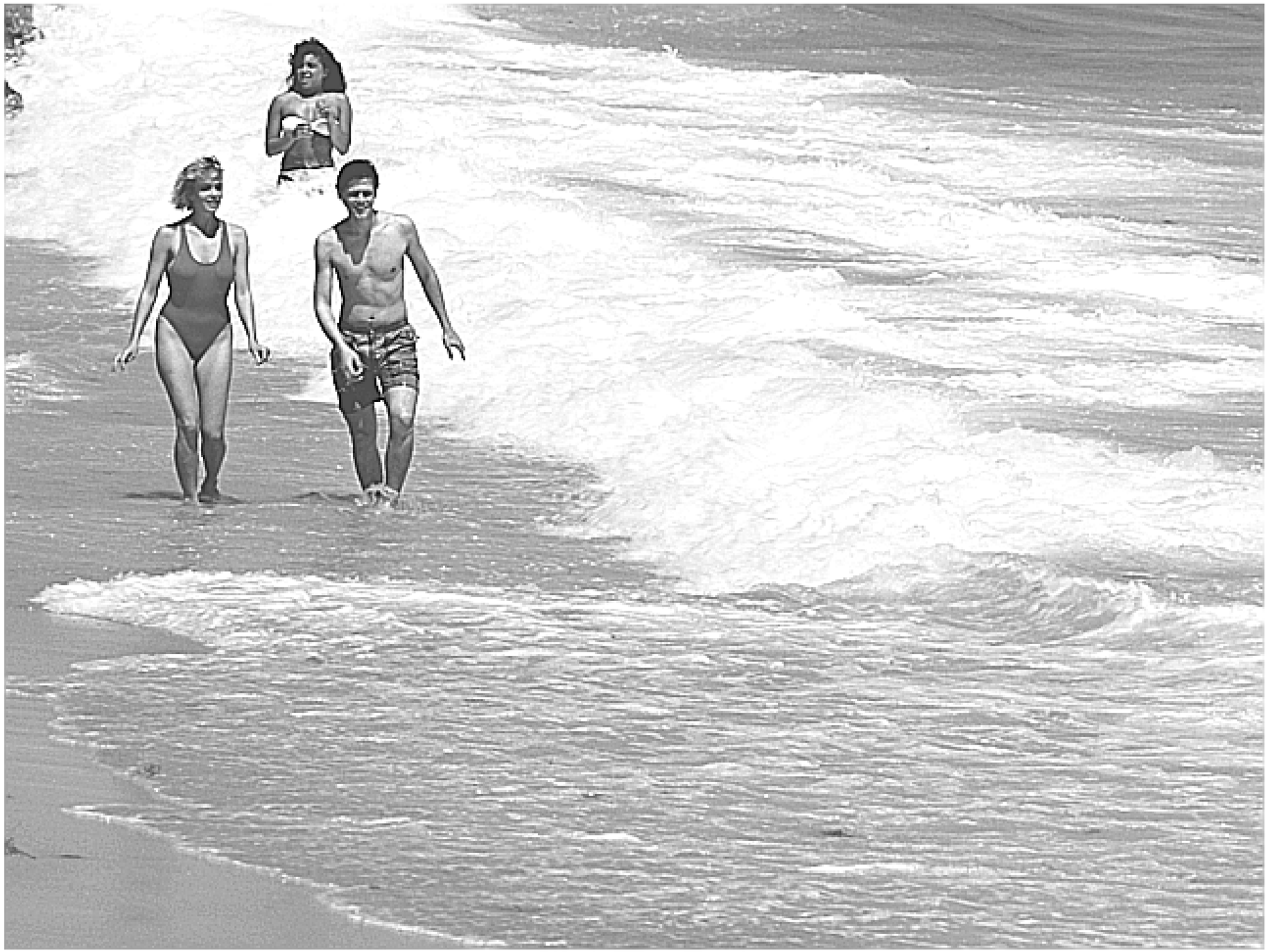
```
Height(I,J) = P(I,J) - Average;
S = 2.0;
P(I,J) = Average(I,J) + S * Height(I,J);
```

The last formula is no longer an equation; it's an assignment statement, because we actually change the value of P(I,J) in the hope that our new image will be more interesting.

ENHANCE: SURF, S = 0.25

ENHANCE: SURF, S = 1.0

ENHANCE: SURF, S = 5.0



ENHANCE: Color Images

Unfortunately, the contrast enhancement operation can be lost when displayed on our rather low-quality projection system! If you are following along on the PDF version of the slides, you should have a better view of what is going on.

Although I have tried so far to avoid color images, let me stray from the black and white path for a moment, and show you how the same contrast enhancement process can improve a great moment in sports:

http://people.sc.fsu.edu/~jburkardt/data/png/underwater_bmx.png

In the first image, I cannot see the spokes of the bicycle wheels. However, the contrast-enhance operation makes the spokes appear clearly. (This is more obvious on the computer screen than over our classroom projector!)

ENHANCE: AQUA BMX, S = 1.0

ENHANCE: AQUA BMX, $S = 5.0$

ENHANCE: AQUA BMX, S = 10.0

ENHANCE: MATLAB Implementation

Here is a simple version of contrast enhancement:

```
s = 2.0;                    <-- or whatever value we want.
p = double ( g );           <-- remember why we do this?
[ m, n ] = size ( g );
p2 = p;                     <-- Good idea for TWO reasons!
for i = 2 : m - 1           <-- Why not 1 : m?
  for j = 2 : n - 1
    average = 0.25 * ( p(i+1,j) + p(i-1,j) ...
                     + p(i,j+1) + p(i,j-1) );
    height = p(i,j) - average;
    p2(i,j) = average + s * height;  <-- Why P2, not P?
  end
end
g = uint8 ( p2 );
```

ENHANCE: Procrustes: One Size Fits All

The mythological innkeeper Procrustes advertised that he had a bed that would fit every customer, short, medium or tall!

Compare and contrast truncation and dilation!

ENHANCE: Rescaling

One thing we have seen before in our image experiments is that we might replace a pixel's value by a number greater than 255 (or less than 0), in which case, the conversion to **uint8** truncates it back to the range [0,255].

The result is a pool of white (or black) where we lose detail.

The real problem is that our new pixel colors strayed outside the legal range. But if we know that's likely to happen, and we really want the "color" 300 to be "whiter than" the color 255, we can recalibrate the colors, that is, rescale them, rather than simply truncating the extreme values.

```
p_max = max ( max ( P ) );
p_min = min ( min ( P ) );
P = 255 * ( P - p_min ) / ( p_max - p_min );
```

This dilation stretches or squeezes our data to fit [0,255]!

Image Algorithms

- Overview
- Representing an Image
- The Portable Gray Map Format
- Image Histograms
- Contrast Stretching
- Contrast Enhancement
- **Salt and Pepper Noise**
- Gaussian Noise
- Edge Detection
- Components
- Projects

SALT: Introduction

Sometimes an image can have a more serious problem than being dark or washed out. The physical process of recording and storing an image is subject to disturbance and damage.

One example occurs in certain recording devices, including satellite scanners, but also regular cameras. What happens is that, for certain pixel positions, the camera fails to record the actual color or shade. Instead, it reports either the highest or lowest possible value.

In a grayscale image, the affected pixels will show up as a scattering of black or white spots, and for this reason, this kind of damage to an image is called salt and pepper noise.

It may seem like a very specialized kind of problem, but it occurs often enough that techniques are needed to deal with it.

SALT: Sample Images

http://people.sc.fsu.edu/~jburkardt/data/png/balloons_noisy.png
http://people.sc.fsu.edu/~jburkardt/data/png/glassware_noisy.png

SALT: Can We Ignore the Noise?

There are several aspects of this problem to keep in mind.

1) An image has a lot of extra information in it; most of an image consists of regions of pixels of roughly the same color.

2) The eye is very sensitive to sudden changes in color or brightness. When noise artificially inserts many such changes, the eye has trouble seeing the "real" picture. Even if 99% of the pixels are good, the eye focuses on the bad ones.

3) The salt and pepper noise means that our smooth regions of roughly equal color will occasionally be interrupted by one extreme and meaningless value. If we use averaging, as we did in contrast enhancement, then we will be diluting the bad pixel somewhat, but it would be better if we could **ignore** it!

SALT: The Median

The average is an attempt to produce one value that fairly represents all the values present.

Since we expect cases where one value is essentially meaningless, we would like to eliminate it from the final result.

Instead of an average, we should use the median, which sorts the data and takes the middle one.

	Average	Median
1,2,3,4,5	3	3
0,8,8,8,8	6.4	8
3,5,5,7,1000	204	5

In MATLAB, **median(v)** returns the median of vector **v**.

SALT: MATLAB Implementation

This algorithm uses a 3x3 neighborhood.

```
p = double ( g );
[ m, n ] = size ( g );
p2 = p;
for i = 2 : m - 1
  for j = 2 : n - 1
    p2(i,j) = median ( ...
      [ p(i+1,j-1), p(i+1,j), p(i+1,j+1), ...
        p(i,  j-1), p(i,  j), p(i,  j+1), ...
        p(i-1,j-1), p(i-1,j), p(i-1,j+1) ] );
  end
end
g = uint8 ( p2 );
```

SALT: Images After 3x3 Treatment

SALT: Summary

The technique we used here is called a median filter.

We didn't really extract any information from the pictures that had salt and pepper noise. At best, we can say we managed to hide some of the false information.

Really, the important thing going on here was that we needed to modify the image in a way that would make it more acceptable to the eye. And that meant, so far as possible, to restore the smooth, slowly changing regions of shade or color, and to ignore or destroy the sudden noisy peaks.

We could make **all** the noise go away in the color photograph by using a 5x5 or 7x7 neighborhood, but as we increase the size of the neighborhood, we increase the work, and we introduce blurriness into the picture.

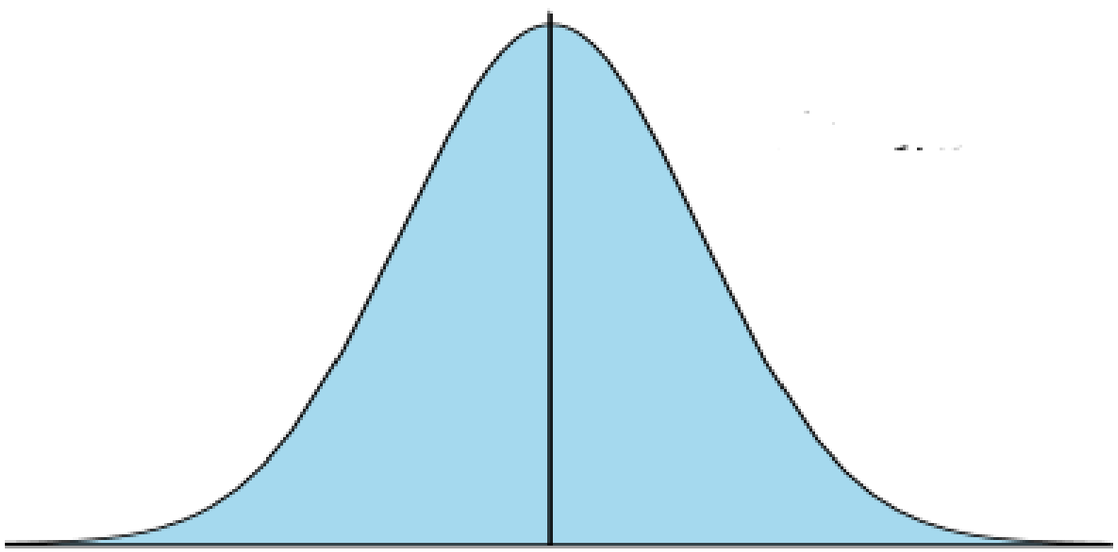
Image Algorithms

- Overview
- Representing an Image
- The Portable Gray Map Format
- Image Histograms
- Contrast Stretching
- Contrast Enhancement
- Salt and Pepper Noise
- **Gaussian Noise**
- Edge Detection
- Components
- Projects

GAUSS: Introduction

Salt and pepper noise was relatively easy to detect and treat. More often, however, every pixel of an image may be more or less distorted by a noisy signal that interferes with the clear, smooth image we are expecting.

The simplest model of the result of this kind of noise uses the Gaussian function or "bell-shaped curve". The idea is that the value in each pixel is close to a correct value, but small errors are very probable, and big errors happen occasionally.



GAUSS: Errors in 2D

Unless we have a perfect camera, the information from a single point in the object we are looking at is likely to wind up not just in a single pixel, but also will affect neighboring pixels somewhat, by blurring.

The typical size of such errors can be described by something called the *standard deviation*, symbolized by σ.

We try to model errors using the Gaussian distribution.

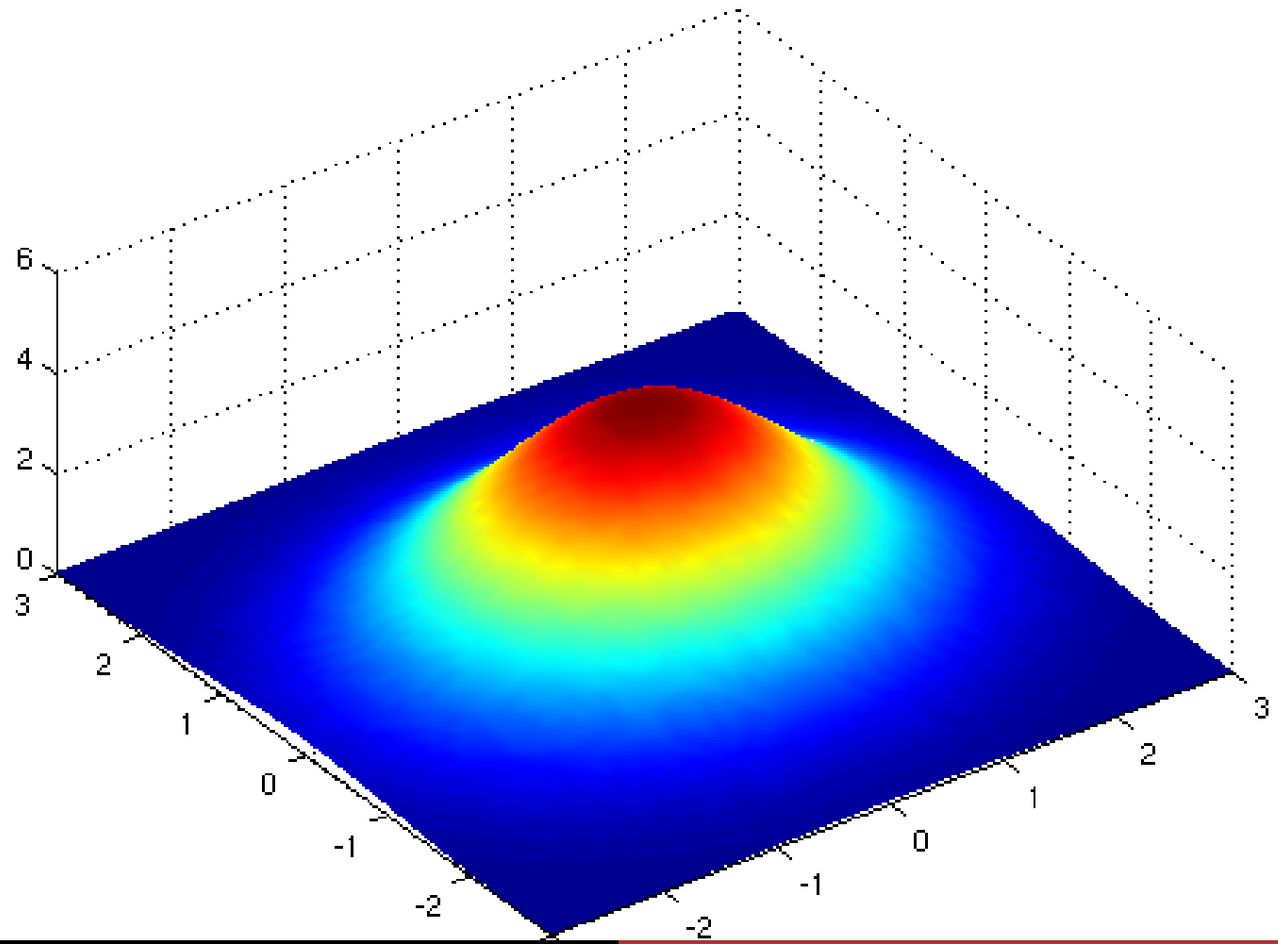
Here is one version of that distribution in 2D:

$$f(x, y) = e^{-\frac{x^2+y^2}{2\sigma^2}}$$

GAUSS: Errors in 2D

Because the true data ends up in more than one pixel, the nearby pixels actually have some information that can be useful. If the central pixel is corrupt, or even if it is missing!, we may be able to recover a good estimate of the correct image value.

The value of σ is unknown, but we can start with 1 and adjust it.



GAUSS: Weighted Averaging

This technique of dealing with noise is called Gaussian averaging or filtering or smoothing.

We choose a neighborhood around each pixel, say a 3x3 or 5x5 block with the pixel at the center. We replace the value at the pixel by a new value computed as a weighted average of the values in the block. The original pixel will dominate the result, but the neighbors will influence its new value.

If the variance is reduced below 1, we are "believing" the central pixel the most. Increasing the variance will allow neighbors to have a stronger say. As we increase sigma, or the size of the neighborhod, the processed picture will go from **sharp/noisy** to **smooth** to **blurry**.

GAUSS: Compute Coefficients

```
function w = gaussian_smoothing ( n, v )

  c = floor ( n / 2 ) + 1;

  for j = 1 : n
    for i = 1 : n
      w(i,j) = exp ( - ( (i-c)^2 + (j-c)^2 ) / (2*v) );
    end
  end

  s = sum ( sum ( w ) )
  w = w / s;

  return
end
```

GAUSS: Example Coefficients

Assuming a variance of 1, here are the 3x3 coefficients:

```
0.0751    0.1238    0.0751
0.1238    0.2042    0.1238
0.0751    0.1238    0.0751
```

and the 5x5 coefficients:

```
0.0030    0.0133    0.0219    0.0133    0.0030
0.0133    0.0596    0.0983    0.0596    0.0133
0.0219    0.0983    0.1621    0.0983    0.0219
0.0133    0.0596    0.0983    0.0596    0.0133
0.0030    0.0133    0.0219    0.0133    0.0030
```

GAUSS: MATLAB Implementation

```
p = double ( g );
[ m, n ] = size ( g );
p2 = p;
w = gaussian_smoothing ( 5, 1.0 );

for i = 3 : m - 2
  for j = 3 : n - 2
    wp = w(1:5,1:5) .* p(i-2:i+2,j-2:j+2);
    p2(i,j) = sum ( sum ( wp ) );
  end
end

g = uint8 ( p2 );
```

GAUSS: 5x5 Gaussian Smoothing

The smoothed image is on the right, although the projector may make that difficult to tell!

http://people.sc.fsu.edu/~jburkardt/data/png/lena_noisy.png

Image Algorithms

- Overview
- Representing an Image
- The Portable Gray Map Format
- Image Histograms
- Contrast Stretching
- Contrast Enhancement
- Salt and Pepper Noise
- Gaussian Noise
- **Edge Detection**
- Components
- **Projects**

EDGE: Automatic Recognition

We have seen that the eye is not very good at seeing the pattern in an image when salt and pepper noise is added. However, there are things the eye can do so automatically that we can't explain how, and we may not even realize that it is being done.

In particular, the eye is extremely good at:

- recognizing edges;
- determining the extent of an object or "component";
- recognizing movement;
- recognizing an object, even when rotated or badly lit;
- recognizing a face, which is never the same twice.

The recognition of edges is one of the simpler tasks on this list.

EDGE: What is an Edge?

An edge is a surprising event!

EDGE: What is an Edge?

We need to think about what we mean by a visual edge, and hope that we can convert this into an operation on the numbers in an image file.

If we are walking in the park and we come to an edge, we are surprised. This suggests that an edge represents a difference between what we expect and what we encounter. Something new is about to occur!

We do not say we have come to an edge if the ground we are walking on slopes downward or upward, as long as this happens at about the same rate.

Just from your experience in walking, you might agree that a linear function is expected and predictable, and that *an edge occurs when linear behavior is disrupted*.

EDGE: Slope Measures Changes

One way to describe linear behavior is to say is has a constant derivative or slope. Another way is to say that if we measure the quantity at equally spaced positions, it increases by the same amount with each step.

We can think about each row or column of our pixel array as equally spaced measurements, and so if we monitor the changes, we can spot the places where a jump seems to occur.

We estimate the "right/left" and "top/bottom" slopes at P(I,J):

```
right/left slope   =   P(I,J+1) - P(I,J-1)
top/bottom slope   =   P(I+1,J) - P(I-1,J)
```

Our data is equally spaced, so it's not so important that we didn't divide by a "Delta X" - or we can take it as 1.

EDGE: Our Edge Detector

A slope of large magnitude, in either direction, says something is changing fast over a short range. This sounds like what we mean by an edge.

So if we take the absolute value of both slopes and add them, we get a measure of how fast things change at each pixel.

```
E = | P(I,J+1) - P(I,J-1) | + | P(I+1,J) - P(I-1,J) |
```

The value of **E** is zero at places where the pixels are "flat" and is large when nearby values differ a lot.

http://people.sc.fsu.edu/~jburkardt/data/png/coins.png

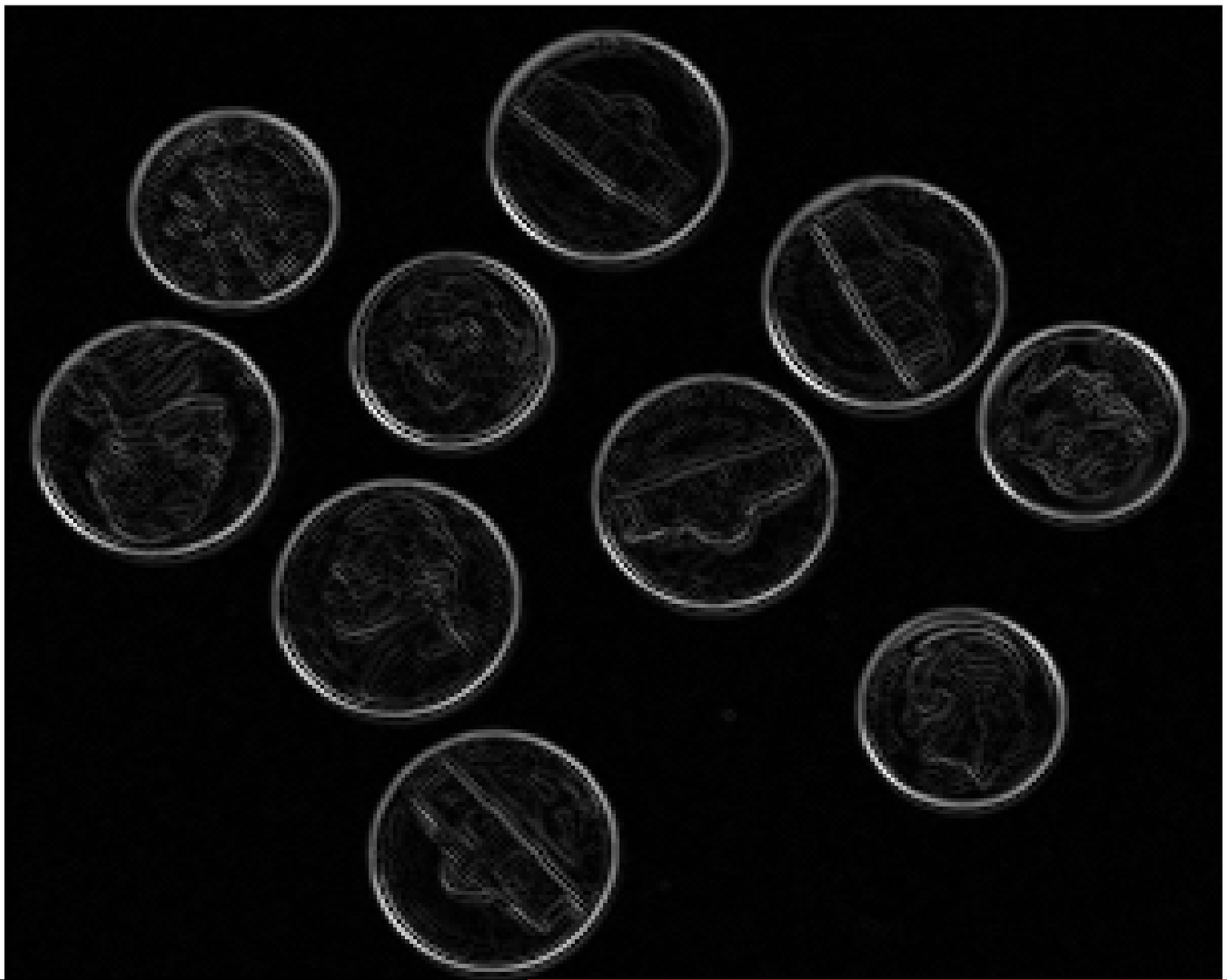
EDGE: A Test Image

EDGE: Read in Data, Compute E

```
a = imread ( 'coins.pgm' );
[ m, n ] = size ( a );

b = double ( a );
e = zeros ( m, n );
e(2:m-1,2:n-1) = abs ( b(3:m,2:n-1) - b(1:m-2,2:n-1) ) ...
               + abs ( b(2:m-1,3:n) - b(2:m-1,1:n-2) );

emin = min ( min ( e ) );     <-- Scale E to [0,255]
emax = max ( max ( e ) );         so we can plot it!
e = round ( 255 * ( e - emin ) / ( emax - emin ) );

imshow ( uint8 ( e ) );
title ( 'All the E data.' );
```

EDGE: The Value of E

All the E data.

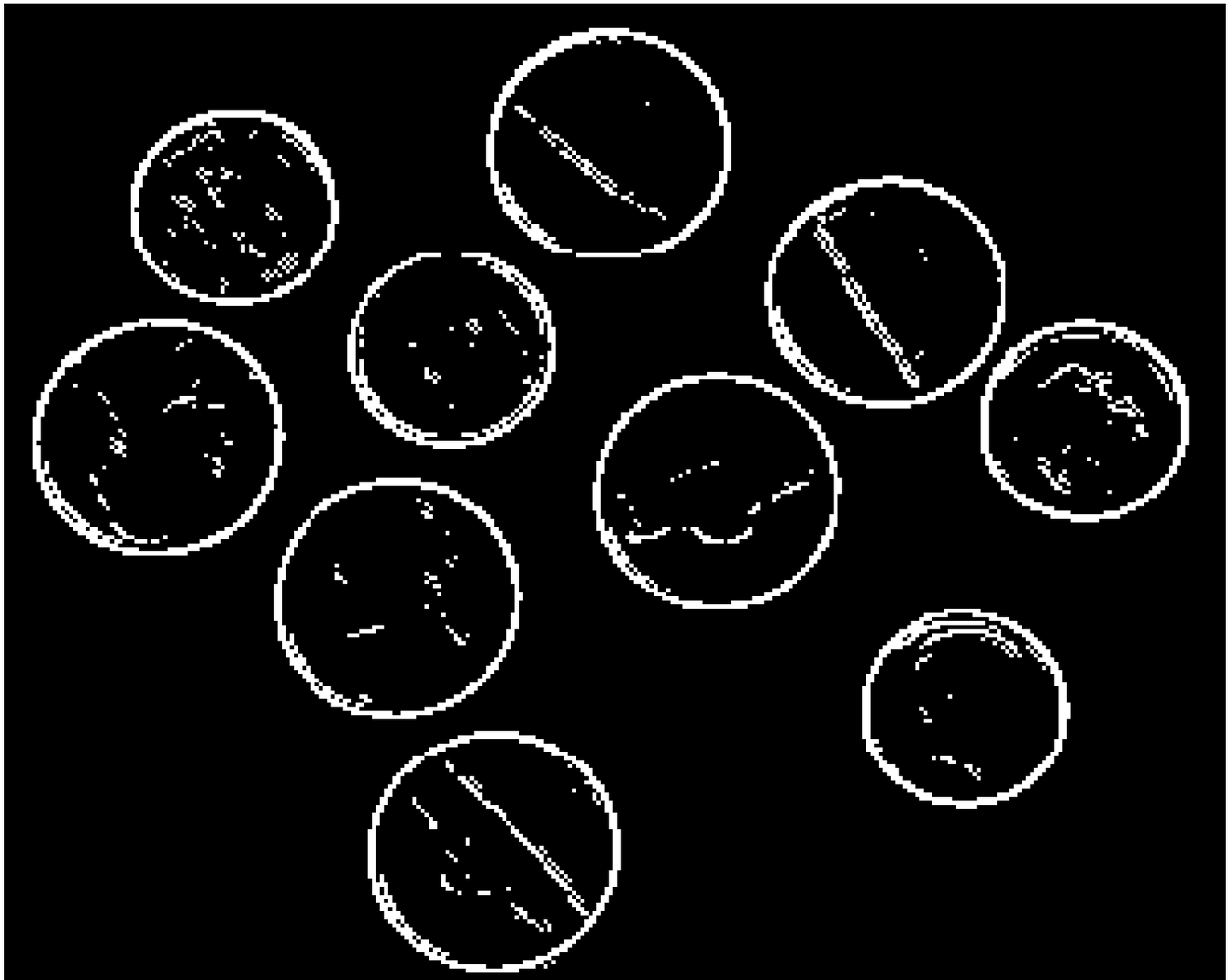


EDGE: Use a Threshold

```
thresh = 50;
e = 255 * ( thresh < e );
imshow ( uint8 ( e ) );
title ( 'E data above the threshold.' );

e_reverse = 255 - e; <-- Reverse video
imshow ( uint8 ( e_reverse ) );
title ( 'E data above the threshold (reverse video).' );
```

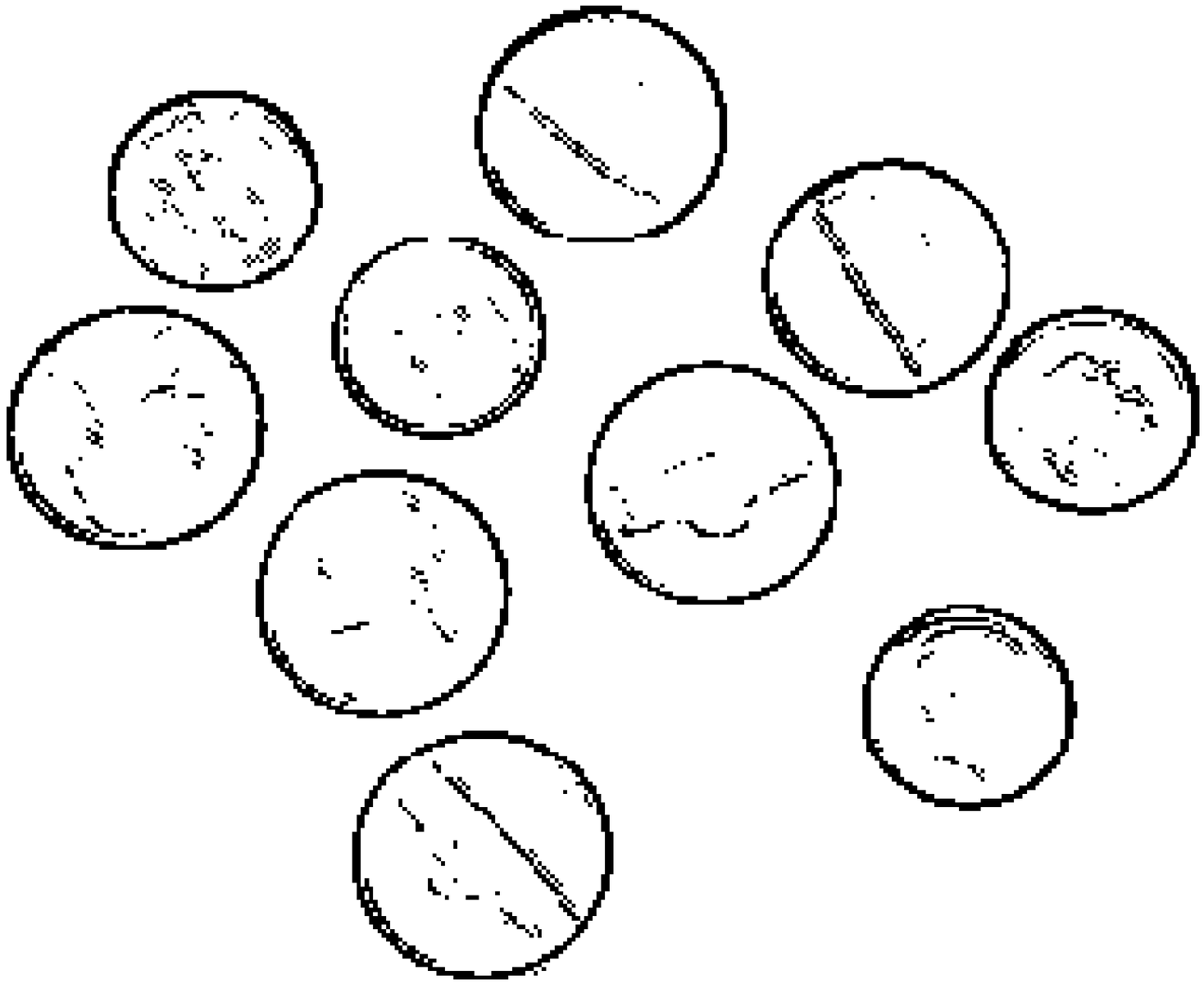
EDGE: E with Threshold

E data above the threshold.



EDGE: E with Threshold and Reverse Video

E data above the threshold (reverse video).



EDGE: Use a Threshold

```
e2 = max ( e(2:m+1,2:n+1), double ( a ) );
a2 = uint8 ( e2 );

r = a2;
g = a;
b = a;

rgb = cat ( 3, r, g, b );

imshow ( rgb );
title ( 'Original gray data, with edges in red.' );
```

EDGE: Coins Plus Edges

Original gray data, with edges in red.

EDGE: Comments

We can call **E**, our edge statistic, the "NEWS" value, since it combines the north, east, west and south values.

Automatic edge detection means a program can search for roads or vehicles in a photograph.

Our formula has some disadvantages, since it only looks for large slopes. A better indicator would check for large *changes* in the slope. This is essentially an estimate for the second derivative. A simple formula for doing this can be constructed out of the NEWS values plus C for the center.

QUIZ: What is the formula to estimate the second derivative for $f(x)$ on equally spaced data? What about $f(x, y)$?

EDGE: Comments

Another thing I should point out is that, with our computation of the quantity **E(*,*)**, we haven't completely "recognized" edges.

A single black pixel in a sea of white will get a high **E** value. But is this an edge?

If a line of pixels all have a high **E** value, then that line will be obvious if we draw it in red, but did we actually realize that these pixels formed a linear edge?

To really recognize edges from image data, we need to carry out more processing on the **E** array, looking for lines and curves that seem to indicate the borders of objects.

Although the algorithm demonstrated here is only a small part of an edge recognition process, I hope you can see how such a process might begin, and what is needed to improve and complete it!

EDGE: Comments

An interesting question that came up in class was this

Suppose your image data was coming from a Wii Kinect, which measures color and distance. Does the extra information help you to detect edges?

It is almost always true that more information is better; certainly it is if you can see a way to take advantage of it.

Here, instead of RGB data, we have RGBD data, that is, color and distance. The D information would allow us to detect an edge between a blue person and a (far away) blue background, which color information alone could not detect, while the color, as before, can catch edges between two objects at the same distance.

EDGE: Comments

I'm afraid that in my example, I snuck in two useful techniques we really haven't talked about.

Thresholding is the method of selecting some range of pixels to preserve, while setting lower values to 0 (black) and higher ones to white. We used thresholding to try to display only those pixels where the E statistic was high.

Reverse Video is obviously the presentation of an image made by interchanging dark and light. For an image of 8-bit depth, we replace each pixel value **P(I,J)** by **255-P(I,J)**. Again, the main reason for this is that the eye is better able to see small black details on a white background than the other way around.

Image Algorithms

- Overview
- Representing an Image
- The Portable Gray Map Format
- Image Histograms
- Contrast Stretching
- Contrast Enhancement
- Salt and Pepper Noise
- Gaussian Noise
- Edge Detection
- **Components**
- Projects

COMPONENTS: Edges

In our first stab at detecting edges, we ended up with a bunch of pixels whose **E** value was high. But it makes a huge difference whether these pixels are scattered, or arranged in lines or layers that can be organized into an edge, or boundary.

To automatically count the coins in the picture, we not only have to detect edges, but group all the pixels that form an edge into a single object.

There are other cases where some pixels in an image are special, but we need to determine whether these special pixels are near each other and form larger objects. The color of the pixel, for instance, might be what we are working from.

COMPONENTS: Edges

What we are trying to do in such cases is to organize the individual pixels into larger objects, that is, to try to detect the large, connected physical objects whose light we are seeing.

This is another example of a process that goes on so quickly and automatically in your brain that you're not aware that it happens, or how hard it is.

We will see how hard it is when we try to export our knowledge, that is, to make some kind of algorithm that will carry out the task for us.

COMPONENTS: Magnetic Resonance Imaging

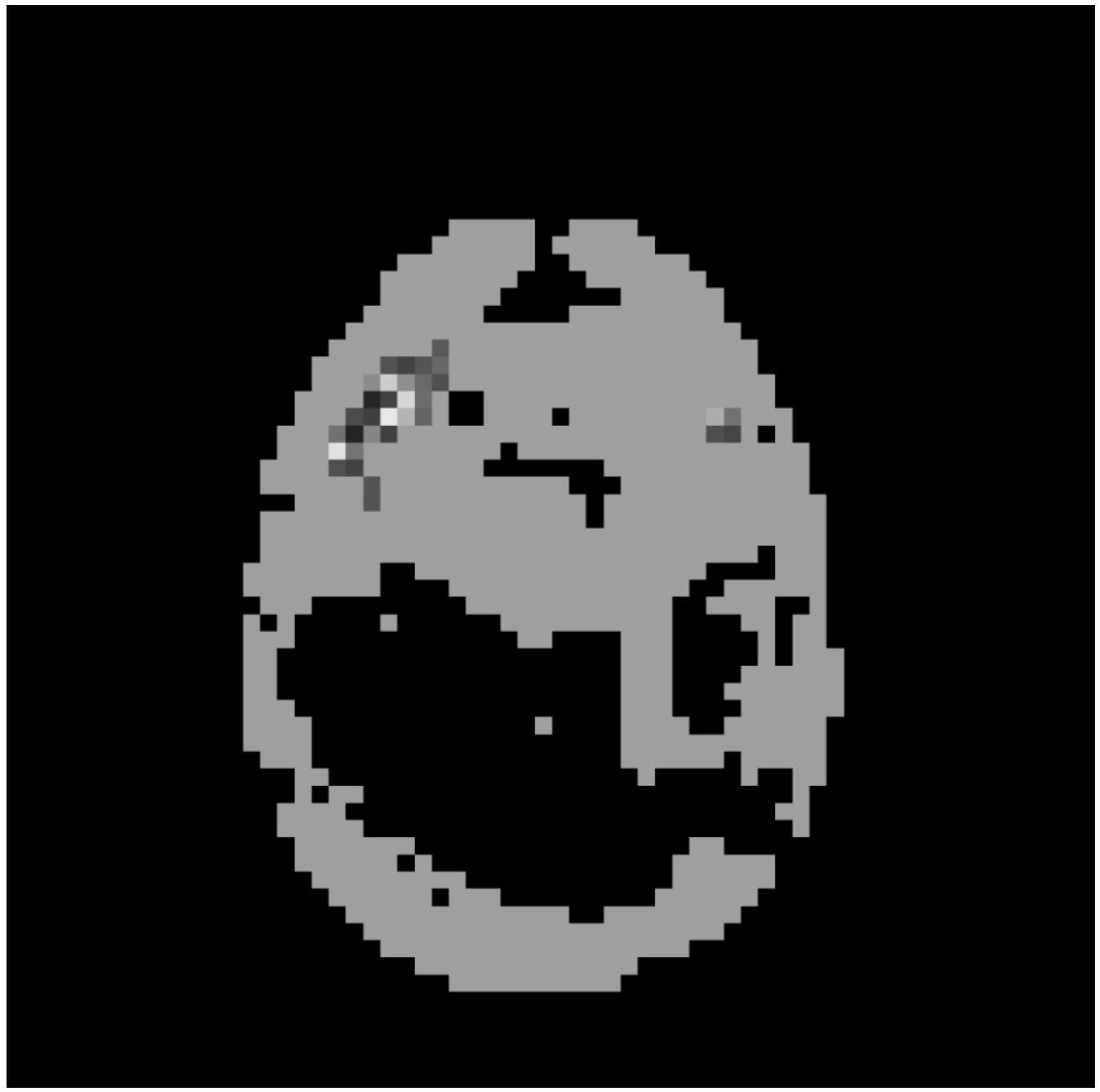
A magnetic resonance imaging system, or MRI, applies a strong magnetic field to a specimen (such as you), and then issues radio frequency signals to measure the response of the body. A kind of scanner systematically records this measurement over a grid of regions, which you can think of as centers of boxes.

There are typical response values for skin, bone, spinal fluid, and brain matter. If tumors are growing in the brain, the affected regions will register a different response value.

Important judgments can be made based on the volume and number of affected regions. Thus, if we know there are 30 boxes that showed tumor response, we still need to know if those 30 boxes are scattered, or form a single, continuous mass.

COMPONENTS: One Slice of an MRI

The very light squares in the northwest region are tumors.



COMPONENTS: Magnetic Resonance Imaging

In the example image, we actually have a 64 x 64 array of integers between 0 and 255. The MRI response is never exact, so we have to estimate that tumors are characterized by a shade of between 200 and 255, for instance.

We can use thresholding to make everything disappear from the picture except for the tumor regions, and reverse video to make them easier to see. Does that answer our question?

Not really! First, while our eye can spot the clusters, we need to do this automatically, with an algorithm.

Secondly, while our eye can see the connections in a 2D slice, what we actually get is a 3D dataset, which is much harder to process.

COMPONENTS: The Algorithmic Challenge

To simplify the discussion, let's assume from now on that our data array contains a 1 for a tumor region and 0 otherwise. This is essentially the same as saying that we've thresholded the data, setting nontumors to black, and then set the tumors to white.

Now we know our task:

Given a 2D or 3D array of 1's and 0's, we need to be able to answer the questions

- how many connected components are there?
- what is the size of each component?

COMPONENTS: Connected Components?

Remember (please!) back in our discussion of graph theory, how we talked about the connected components of a graph.

Essentially, we noticed that the graph fell apart into components, so that all the nodes in a component were connected to each other, but not to nodes in another component.

Well, this is the same thing, although our graph is implicit. Without drawing edges, we naturally think of a pixel as being "connected" to the 4 pixels that touch it (or 6 in 3D!)

So now we have a special case of the graph connected component problem. It's special because the data is laid out in a rectangular grid. And that's going to make it *much much easier* to analyze.

COMPONENTS: The 1D Case

To solve a hard problem, it helps to warm up on a simple one.

Let's ask how to solve the connected component problem in 1D, (even though we'll never win a prize for doing so!)

Here is a typical data array:

```
0 1 1 1 0 0 0 1 0 0 1 1 0 0 0 0 0 0 1 0 1
```

How many components? Easy: 5!

Component sizes? Easy: 3, 1, 2, 1, 1!

...but could we have answered these questions automatically?

COMPONENTS: The 1D Case

Before we sneak a look at my answer to this problem, let's try to reason it through, and code up an algorithm. This version of the problem is simple enough that we have a chance!

Our input array of **N** pixels **P**:

0 1 1 1 0 0 0 1 0 0 1 1 0 0 0 0 0 0 1 0 1

How do we count components and measure sizes?

```
function [ l, label ] = components ( n, p )
```

L will be the number of components,
LABEL labels each pixel with its component.

———Thinking time!———

COMPONENTS: Algorithm for 1D Case

```
L = 0;                          <-- Use L to label components.
POLD = 0;                       <-- POLD is the previous pixel.

for I = 1 : N
  if ( P(I) == 0 )              <-- P(I) is 0, so ignore it.
    LABEL(I) = 0;
  else                          <-- P(I) is 1, so we need a label.
    if ( POLD == 0 )            <-- Did we start a new component?
      L = L + 1;
    end
    LABEL(I) = L;               <-- Slap on the label!
  end
  POLD = P(I);
end
```

COMPONENTS: The 1D Case

How would this have worked with our example data?

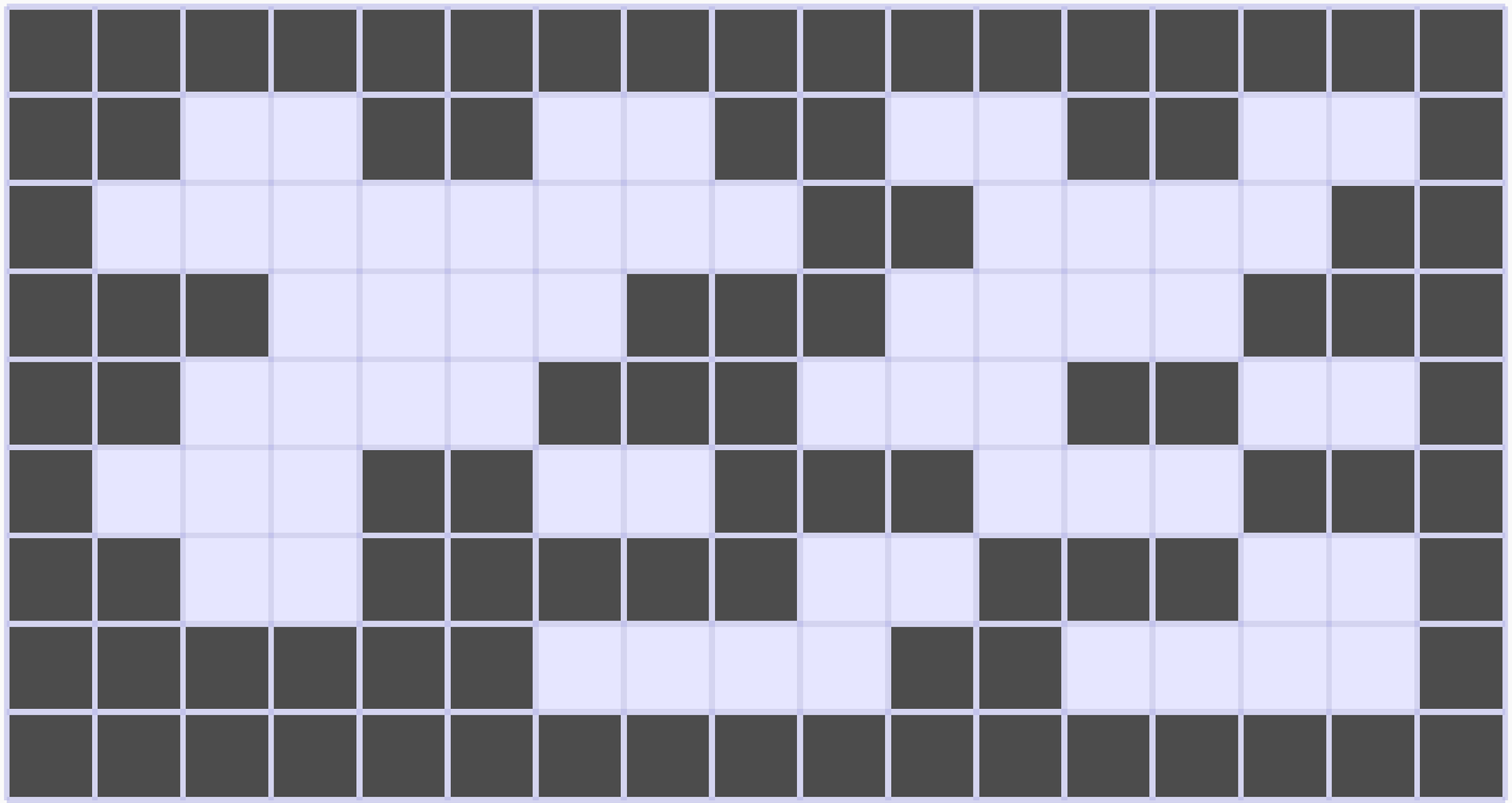
Let's walk through this problem, and fill in the LABEL array:

```
I:       1 2 3 4 5 6 7 8 9101112131415161718192021
P:       0 1 1 1 0 0 0 1 0 0 1 1 0 0 0 0 0 0 1 0 1 -
POLD:    - 0 1 1 1 0 0 0 1 0 0 1 1 0 0 0 0 0 0 1 0 1
LABEL:   _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
```

Can you see how this is a special case of the connected component problem for graphs, but that the fact that this graph is connected like a string of beads makes it extremely simply to simply walk from left to right and do the necessary work?

Some of these same ideas will still work in 2D, if we generalize them a bit, although we will have a few loose ends to deal with.

COMPONENTS: Sample 2D Problem



COMPONENTS: Sample 2D Problem

Even though our problem is two-dimensional, our algorithm will need to walk through the data linearly. Let's suppose that we do this by taking one row at a time.

I already see some problems!

What if the last pixel of one row and the first pixel of the next row are nonzero?

How do we deal with the fact that a pixel can be related to pixels above it as well as to those to the left of it?

COMPONENTS: Sample 2D Problem

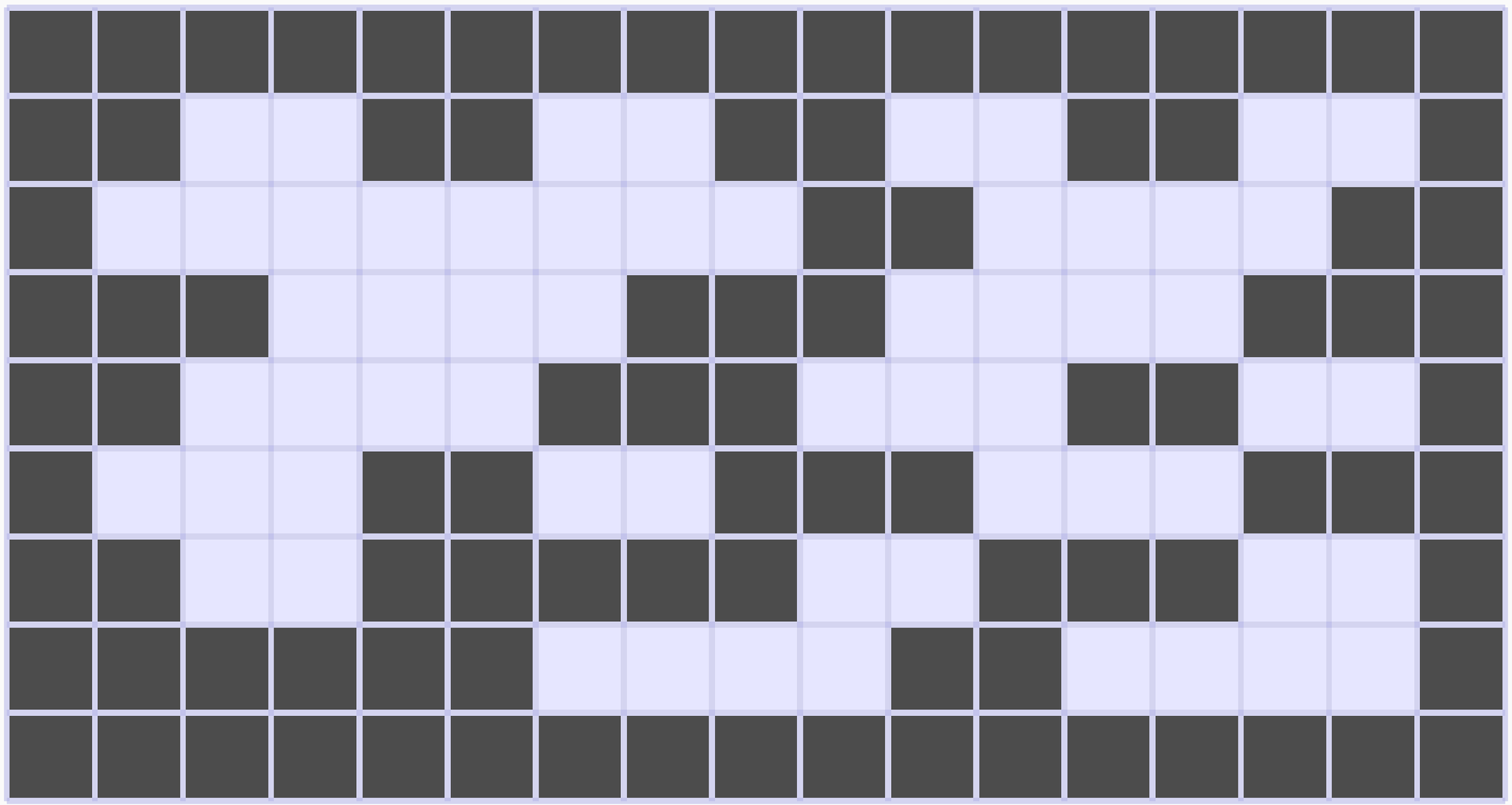
Our first attempt at an algorithm can simply walk through the array, one row at a time, lookup "backwards" (previous column) and "upwards" (previous row) to look for nonzero entries.

If either ancestor is nonzero, then this pixel inherits that component label.

Hmmm, that's going to do the connecting OK, but I can still see a problem!

COMPONENTS: Sample 2D Problem

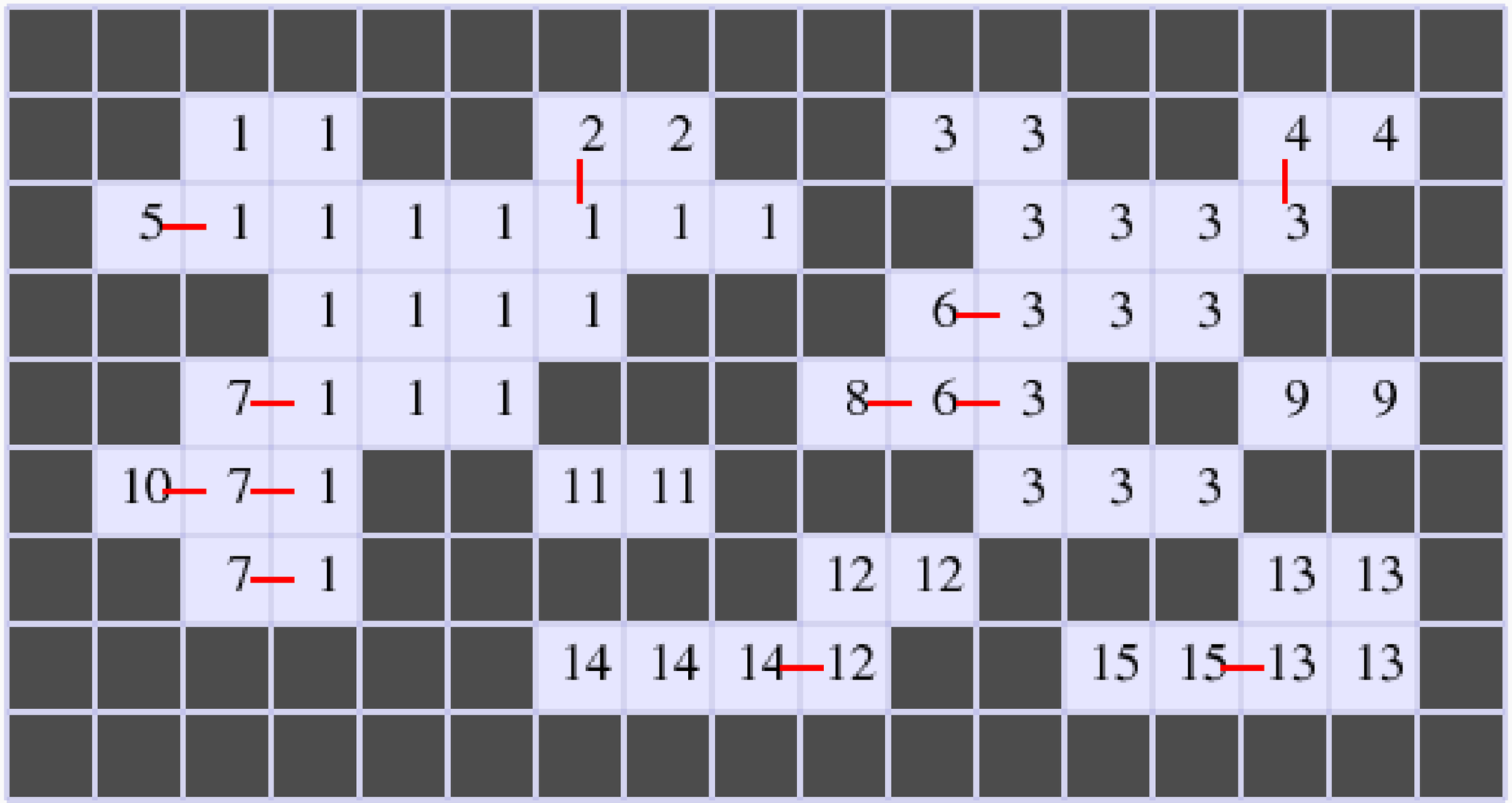
Let's apply our ideas to the sample problem!



COMPONENTS: Algorithm for 2D Case?

```
L = 0;
for I = 1 : M
  for J = 1 : N
    if ( P(I,J) == 0 )
      LABEL(I,J) = 0;
    elseif ( P(LEFT) == 0 and P(ABOVE) == 0 )
      L = L + 1;
      LABEL(I,J) = L;
    elseif ( P(LEFT) == 0 )
      LABEL(I,J) = LABEL(ABOVE);
    elseif ( P(ABOVE) == 0 )
      LABEL(I,J) = LABEL(LEFT);
    else
      LABEL(I,J) = min ( LABEL(LEFT), LABEL(ABOVE) );
    end
end
```

COMPONENTS: After Sweep



COMPONENTS: Label Adjustment

Our algorithm is trying to work, but it is liable to mislabel pixels. If we have a pixel with two nonzero neighbors, left and above, and those labels don't match, then we have a problem. The problem is indicated by the red lines on the graph, which show connected pixels with different labels.

Luckily, every time this problem occurs, we know about it (because a pixel reports two neighbors with different labels) and the cure is to realize that one label should be replaced by the other. For instance, once we realize that "1" and "5" are both being used to label the same component, we need to remember that later on, we must replace all the "5" labels by "1"s.

It actually can get more complicated than that, as you see where 10's should become 7's, but 7's should become 1's... so really 10's should be 1's too!

COMPONENTS: Label Adjustment

The good news is, while a component might have multiple labels, a label never refers to two different components. If we simply choose one label for each component, we can sort out the problem.

We can handle this by keeping an **INDEX** array, which records the adjustments we make to each label.

As long as we use the rule that when given two labels, we keep the lower one, we can figure things out at the end

In the following table, we show the 15 labels used in the diagram, and for each label, INDEX indicates whether we encountered a lower label in the same component. By walking through this array, we can find the minimum label for each component.

Here we only need 6 labels, so we have a FINAL array that assigns "1" through "6" to the pixels based on their original labels.

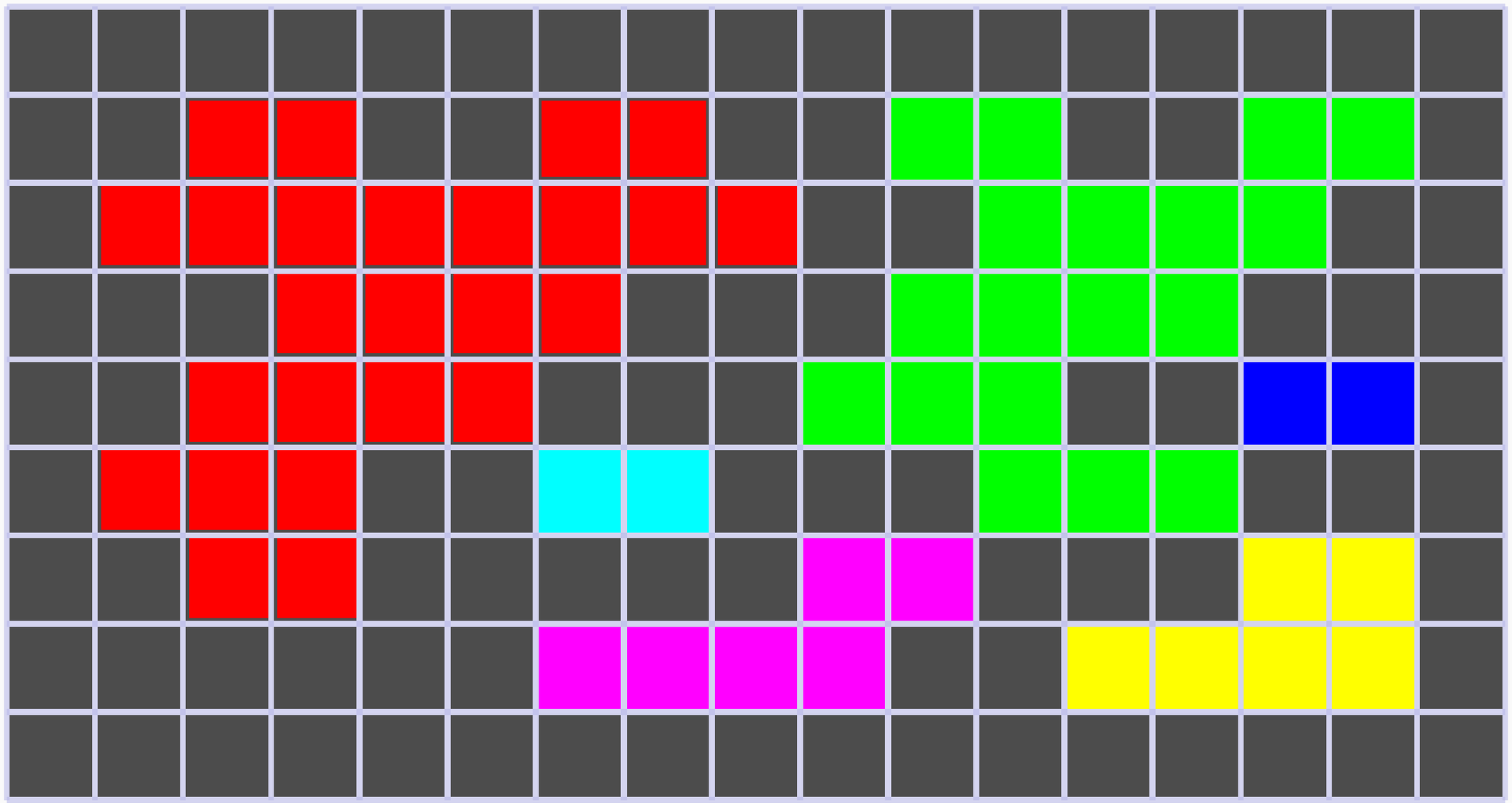
COMPONENTS: Label Adjustments

LABEL	INDEX		UNIQUE	LABEL	FINAL
1 ==	1.		1 1	1 ==>	1
2 ==>	1.			2 ==>	1
3 ==	3.		3 2	3 ==>	2
4 ==>	3.			4 ==>	2
5 ==>	1.			5 ==>	1
6 ==>	3.			6 ==>	2
7 ==>	1.			7 ==>	1
8 ==>	6	(==> 3.)		8 ==>	2
9 ==	9.		9 3	9 ==>	3
10 ==>	7	(==> 1.)		10 ==>	1
11 ==	11.		11 4	11 ==>	4
12 ==	12.		12 5	12 ==>	5
13 ==	13.		13 6	13 ==>	6
14 ==>	12.			14 ==>	5

COMPONENTS: After Label Adjustment

		1	1			1	1			2	2			2	2	
	1	1	1	1	1	1	1	1			2	2	2	2		
			1	1	1	1				2	2	2	2			
		1	1	1						2	2	2			3	3
	1	1	1			4	4				2	2	2			
		1	1						5	5				6	6	
						5	5	5	5			6	6	6	6	

COMPONENTS: Color by Label



COMPONENTS: The 3D Problem

Now that we've seen algorithms for the 1D and 2D cases, you should have some idea of a possible way to attack the 3D case:

Starting at voxel (1,1,1), "count" through the voxels:

```
(1,1,1), (1,1,2), ..., ( 1, 1,nz),
(1,2,1), (1,2,2), ....,( 1, 2,nz),
...      ...      ...,  ( 1,ny,nz),
(2,1,1), (2,1,2), ...,  ( 1, 1,nz),
...      ...            (nx,ny,nz).
```

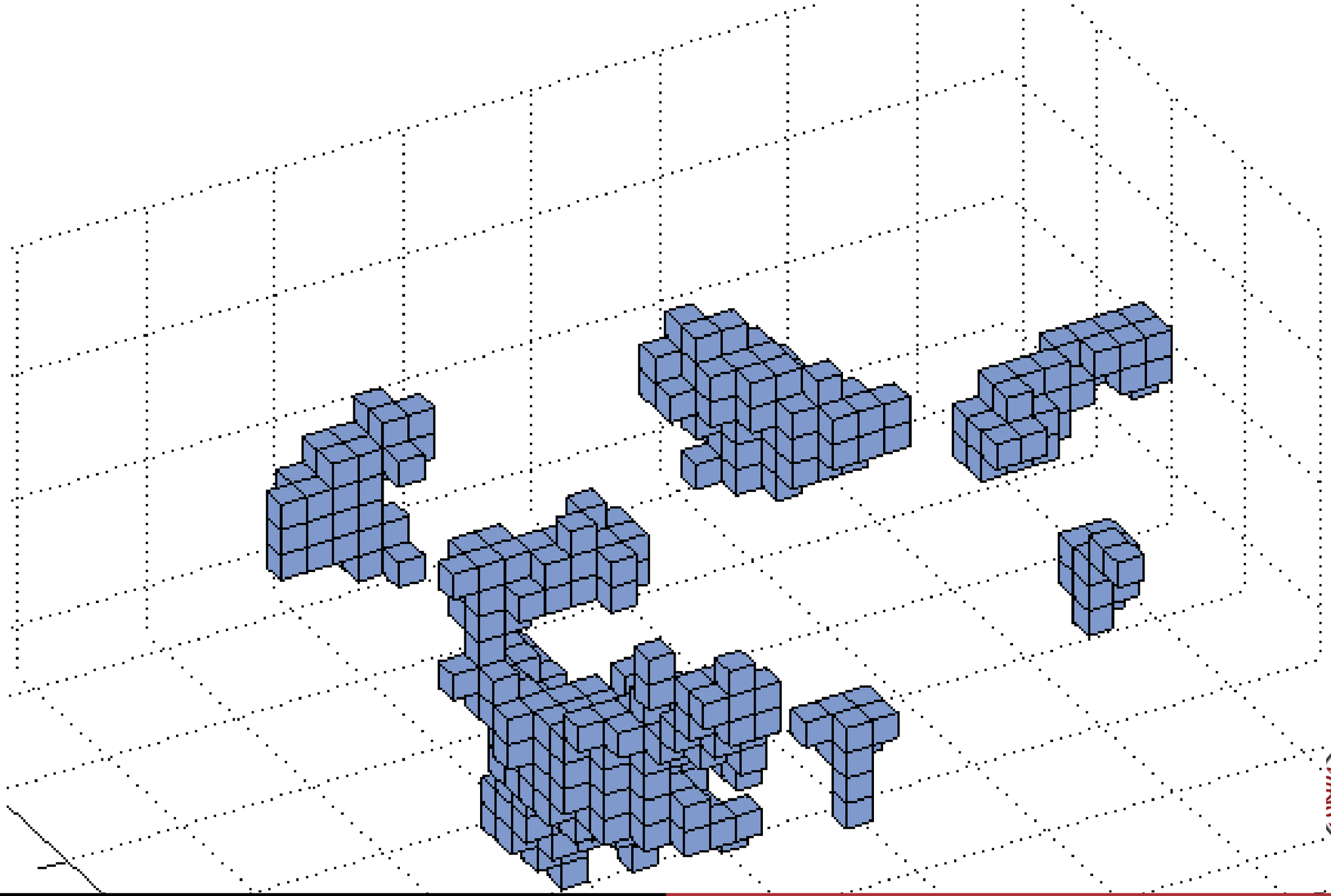
COMPONENTS: The 3D Problem

When you encounter a nonzero voxel:

- If no back neighbors, increment L and use that to label this voxel;
- Otherwise, label this voxel with the minimum of the labels of the three back neighbors;
- If this voxel has multiple back neighbors with different labels, make sure that all labels "point" to the minimum label;

Once you have assigned the labels, walk through the label array looking for unique labels, and determine a relabeling for each label. Use the unique labels to label the components.

COMPONENTS: The 3D Problem (MRI Data)



COMPONENTS: Summary

The components algorithm suggests one way in which a computer vision system can mimic the brain's ability to take what the eye sees (spots of light) and organize it into patterns (stop signs, armadillos, pickles).

Because we essentially number the pixels or voxels, we can choose any component by number and easily change its position, size or color within the image. We could also hide that component or move it to a separate image all by itself.

In cases of medical photographs or 3D MRI scans, we can automatically determine the number of components and their sizes. This could be used to determine the volume of a tumor, or the number of separate tumors.

Image Algorithms

- Overview
- Representing an Image
- The Portable Gray Map Format
- Image Histograms
- Contrast Stretching
- Contrast Enhancement
- Salt and Pepper Noise
- Gaussian Noise
- Edge Detection
- Components
- **Projects**

Projects

- Archimedes Palimpsest: combining multiple images;
- How “false color” images are made in astronomy;
- Analyzing LANDSAT satellite photos;
- Compression of Images;
- Compression of Animated Images on Television;
- Optical Character Recognition (OCR).