Image Processing

http://people.sc.fsu.edu/~jburkardt/isc/week12_lecture_22.pdf

.....

ISC3313:
Introduction to Scientific Computing with C++
Summer Semester 2011

.....

John Burkardt
Department of Scientific Computing
Florida State University

Last Modified: 26 July 2011

Image Processing

- **Introduction**
- Working with Files
- Example Images
- The Portable Gray Map Format
- The Noise Problem
- The Edge Problem
- Lab Exercise #11

INTRO: Schedule

Next Class:

- Detection and Disabling of Explosive Devices

Assignment:

- Today: in-class lab exercise #11

Evaluation:

- Go to http://campus.fsu.edu/esussai and log in. The evaluation is anonymous.

INTRO: One Week Left For Projects!

Please remember that your project is due on Tuesday, August 2nd, at class time, 11am! This includes:

- a 5 minute oral presentation;
- a 3-5 page report to be turned in;
- a C++ program, to be turned in.

If you do not present your talk and turn in your report and programs on time, you will not receive a grade for the course.

INTRO: Images are Scientific Data

In the same way that linear algebra was developed to deal with scientific data stored in vectors and matrices, the new field of digital image analysis and processing is appearing, to help us to convert the data stored in images into information we can use.

In this class, I have already referred to some simple image processing tasks, such as reading in an image of the Mona Lisa, lightening the image, and drawing a few lines on it. This was intended to convince you that an image is simply an array of numbers that your C++ program can modify.

Today, we will look at some simple versions of useful operations on images. Professional image processing uses much more sophisticated techniques. But we will be able to see how to tu statements about images into operations on numbers, so tha C++ can carry out the operations or analysis that we desire.

Image Processing

- Introduction
- **Working with Files**
- Example Images
- The Portable Gray Map Format
- The Noise Problem
- The Edge Problem
- Lab Exercise #11

FILES: C++ Input/Output Channels

A C++ program communicates with the user through three special channels:

- *standard input*, which is normally the keyboard. Statements such as **cin** receive their data here;
- *standard output*, which is normally the terminal screen. Statements such as **cout** send their output here;
- *standard error*, which is the terminal screen. Statements such as **cerr** send their output here.

FILES: Unix Redirection

On Unix systems, it is possible to use redirection symbols so that standard input comes from a file instead of the keyboard:

```
sort < my_albums.txt
```

or to have the standard output redirected to a file

```
fortune > my_future.txt
```

or even both:

```
calculator < my_grades.txt > my_average.txt
```

FILES: Disadvantages

Redirection is not always the best way to work with files:

- In particular, a system like NetBeans might not offer you the option of running the program from a command line, so you can't specify alternative input or output channels;
- On an operating system like Microsoft Windows, programs are usually run by double clicking on an icon, or from some interactive development environment, and the file redirection option is not available.
- If your program should read from several files, or create several files, you can't do that with redirection;
- If you want to have input from a file, but you also want to type some input interactively, redirection is the wrong ch

FILES: Creating Your Own Output File

Let's say that we have a program **table_redirect.cpp** which computes the squares of the numbers up to 10 to make a table:

```
# include <cstdlib>
# include <iostream>
using namespace std;

int main ( )
{
  int i;

  for ( i = 0; i<= 10; i++ )
  {
    cout << "  " << i << "  " << i * i << "\n";
  }
  return 0;
}
```

We could save this table to a file by the commands:

```
g++ table_redirect.cpp
mv a.out table_redirect
table_redirect > table.txt
```

FILES: Creating Your Own Output File

We can create the table file without redirection, using tools from C++. This requires us to:

- **include** <**fstream**>;
- create a variable of type **ofstream** called **table**;
- use the **open** statement so that **table** points to the file *table.txt*;
- replace "**cout** <<" by "**table** <<", sending information directly to the file;
- use the **close** statement to close the file.

FILES: Creating Your Own Output File

table_direct.cpp:

```
# include <cstdlib>
# include <iostream>
# include <fstream>                                  <-- required include statement
using namespace std;

int main ( )
{
  int i;
  ofstream table;                                    <-- Lets us refer to the file.

  table.open ( "table.txt" );                        <-- Creates the file.

  for ( i = 0; i<= 10; i++ )
  {
    table << "  " << i << "  " << i * i << "\n";     <-- Sends data to file.
  }

  table.close ( );                                   <-- Closes the file.
  return 0;
}
```

When we run this program, it creates the file *table.txt* automatically, and stores the data there. It will work the same way from the command line, from NetBeans, or on a Windows machine.

FILES: Creating Your Own Input File

If we wanted to average football attendances, we might write **average_redirect.cpp** to read the data from input:

```
# include <cstdlib>
# include <iostream>
using namespace std;

int main ( )
{
  int attendance, average = 0, i, n = 0;

  while ( true )
  {
    cin >> attendance;
    if ( cin.eof ( ) )
    {
      break;
    }
    n = n + 1;
    average = average + attendance;
  }
  cout << "  Average = " << average / n << "\n";
  return 0;
}
```

And this program could read the data by

```
g++ average_redirect.cpp
mv a.out average_redirect
average_redirect < fsu_football_2010.txt
```

FILES: Creating Your Own Output File

We can read the football attendances file without redirection:

- **include** <**fstream**>;
- create a variable of type **ifstream** called **attend**;
- use the **open** statement so that **attend** points to the file *fsu_football_2010.txt*;
- replace "**cin** >>" by "**attend** >>", reading information directly from the file;
- use the **close** statement to close the file.

FILES: Creating Your Own Input File

```
# include <cstdlib>
# include <iostream>
# include <fstream>                              <-- The include file
using namespace std;

int main ( )
{
  int attendance, average = 0, i, n = 0;
  ifstream attend;                               <-- How we refer to the file

  attend.open ( "fsu_football_2010.txt" );       <-- Open the file

  while ( true )
  {
    attend >> attendance;                        <-- Read from the file
    if ( attend.eof ( ) )                        <-- Did we run out of input?
    {
      break;
    }
    n = n + 1;
    average = average + attendance;
  }
  attend.close ( );                              <-- Close the file
  cout << "  Average = " << average / n << "\n";
  return 0;
}
```

average_direct.cpp accesses the input file directly.

FILES: Summary of Files

If you are able to use Unix redirection, you can put off learning about direct access to files for a while, but eventually it becomes necessary.

You can have several files open at the same time, whether for reading or writing.

It is also possible to open an existing file, and add more information to the end of it.

The files we have considered are text files, which means you could create them with an editor, or print them out. Many computer files are not printable; they are called *binary files*, and use a compressed format for storing information. Many graphics files, for instanc are binary files. C++ includes the ability to read and write b na files, but we will not go into this topic!

Image Processing

- Introduction
- Working with Files
- **Example Images**
- The Portable Gray Map Format
- The Noise Problem
- The Edge Problem
- Lab Exercise #11

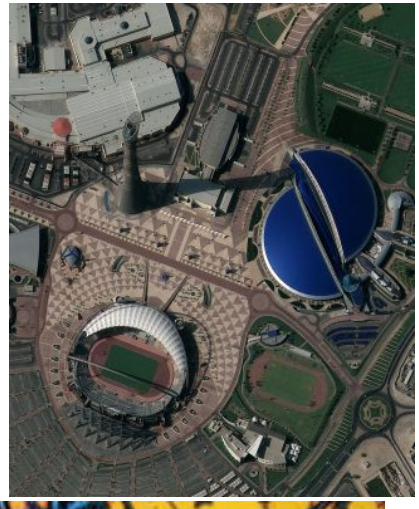
EXAMPLE: Images are Scientific Data

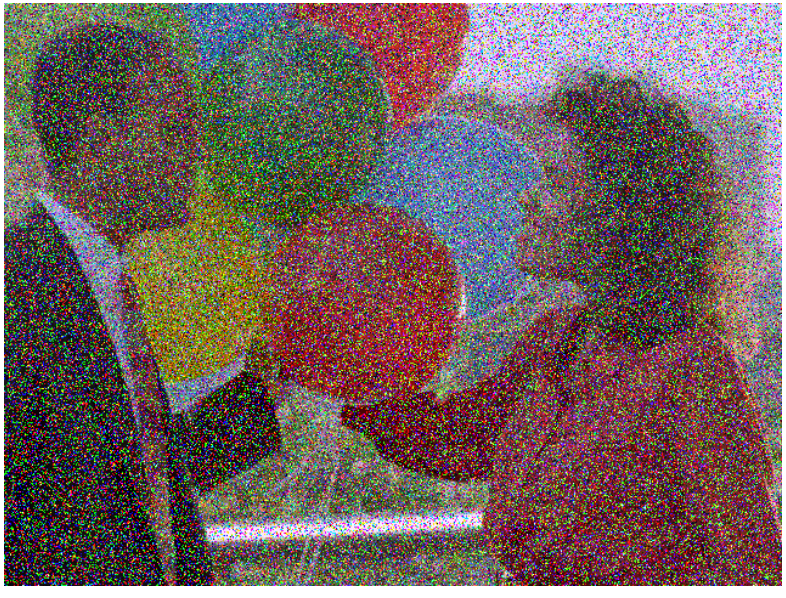
While beautiful images hang in museums, it's important to realize that images also represent a primary form of scientific data.

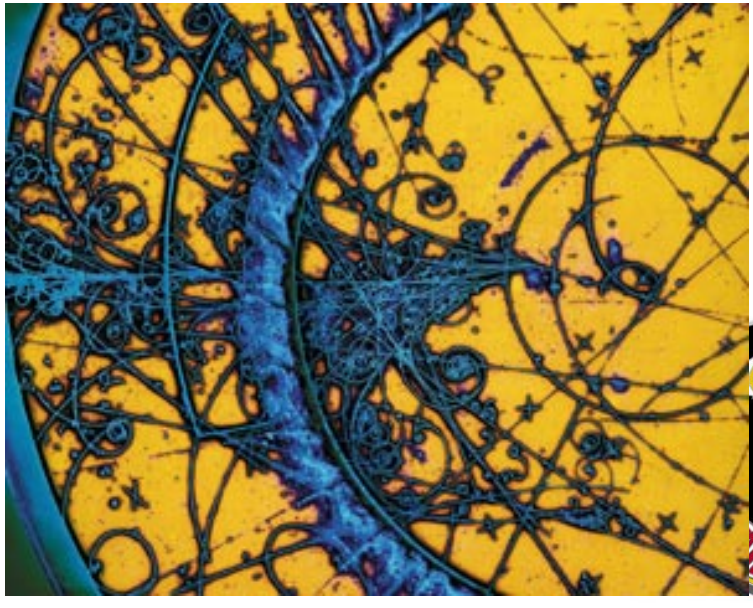
Images are used to store

- astronomical observations;
- medical examinations (X-rays, MRI's, CAT scans);
- particle physics experiments;
- satellite imagery (Google Maps, military observation, agricultural monitoring);
- facial recognition, airport scanners

EXAMPLE: Sample Images







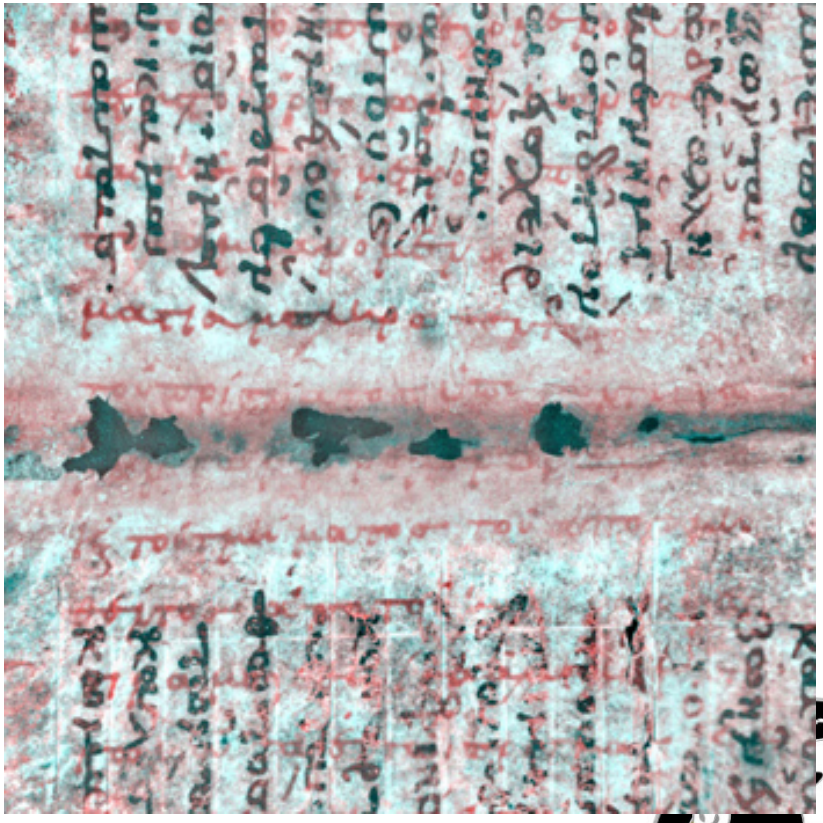
EXAMPLE: Sample Images

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

1. Can we count the stars in the image?
2. What size are the 3D "blobs" in the MRI;
3. A satellite reconnaissance photo;
4. Can we remove the speckles from the photo?
5. Can we detect the kind of particle decay in this collider ?

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

EXAMPLE: Sample Images



EXAMPLE: 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

Image Processing

- Introduction
- Working with Files
- Example Images
- **The Portable Gray Map Format**
- The Noise Problem
- The Edge Problem
- Lab Exercise #11

PGM: Common Image Formats

Image data is typically stored in a file. The file extension often indicates the particular format being used:

- **bmp**, Microsoft Bit Map;
- **gif**, once popular for web graphics;
- **jpg**, what comes out of your digital camera;
- **pbm**, Portable Bit Map (black/white);
- **pdf**, usually for documents;
- **pgm**, Portable Gray Map (shades of gray);
- **png**, an open-source replacement for GIF;
- **ppm**, Portable Pixel Map (RGB color);
- **ps**, usually for documents;
- **tif**, a high-quality photographic format.

PGM: Common Image Formats

On our Linux system, most of these image files can be viewed simply by double clicking the image's icon, or, if you are using a terminal window, you can use the **eog** program:

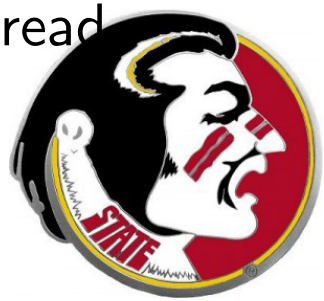
```
eog satellite_photo.png
```

PGM: Common Image Formats

However, every one of these formats differs in the rules for how it organizes information. This means that a C++ program that wants to work with image data stored in a file must know the rules for that file format.

If we are working with a gray scale image, then the program simply wants to set up a 2D array of the appropriate size, perhaps containing integers between 0 and some maximum value.

The simplest way to deal with the variety of image formats is to use a conversion program that turns any file into a single preferred format. Then your C++ program only has to know how to read and write that format.



PGM: The ImageMagick "convert" Program

ImageMagick provides a free program called **convert**, which can convert between over 100 image file formats using a simple command.

For example, to make a Microsoft BMP version of *satellite_photo.png*, we type

```
convert satellite_photo.png satellite_photo.bmp
```

ImageMagick knows what to do based on the file extension.

More information at: **www.imagemagick.org**

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).

There is a related PBM format for black/white images and a PPM format for color.

PGM: The FEEP Example Displayed

The nice thing about the ASCII PGM format is that, for a very small file, you can almost see the picture. For instance, here is a zoomed-in look at an image, using 24 columns and 7 rows of pixels, of the letters **FEEP**

PGM: The FEEP Example in a PGM File

Our data is stored as an ASCII PGM file **feep.ascii.pgm**:

```
P2       <-- Indicates this is an ASCII PGM file
24 7     <-- There are 24 columns and 7 rows
15       <-- The maximum gray is 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
```

PGM: Read the FEEP PGM File Into a 2D Array

A simple PGM file can be read by simple C++ commands. Assuming we know in advance that our image data is going to be 24 columns and 7 rows, we can write **feep_read.cpp**:

```
char c1, c2;
int cols, i, j, maxg, p[7][24], rows;

cin >> c1 >> c2;
cin >> cols >> rows;
cin >> maxg;
for ( i = 0; i < rows; i++ )
{
  for ( j = 0; j < cols; j++ )
  {
    cin >> p[i][j];
  }
}
```

PGM: The FEEP Example in a PGM File

If you don't actually know the size of the image beforehand, you can still read the PGM file, but you have to wait until after you have gotten the number of rows and columns in order to set aside the space.

Moreover, we have to store the **p** data as a 1D array, not a 2D array!

PGM: Read ANY ASCII PGM File into a 1D Array

```
char c1, c2;
int cols, i, j, maxg, rows;
int *p;                              <-- P is an int array whose
                                         size isn't decided yet.

cin >> c1 >> c2;
cin >> cols >> rows;
cin >> maxg;
p = new int[rows * cols];    <-- P uses rows * cols ints
for ( i = 0; i < rows; i++ )
{
  for ( j = 0; j < cols; j++ )
  {
    cin >> p[i*cols+j];      <-- Access p[i][j] this way.
  }
}
```

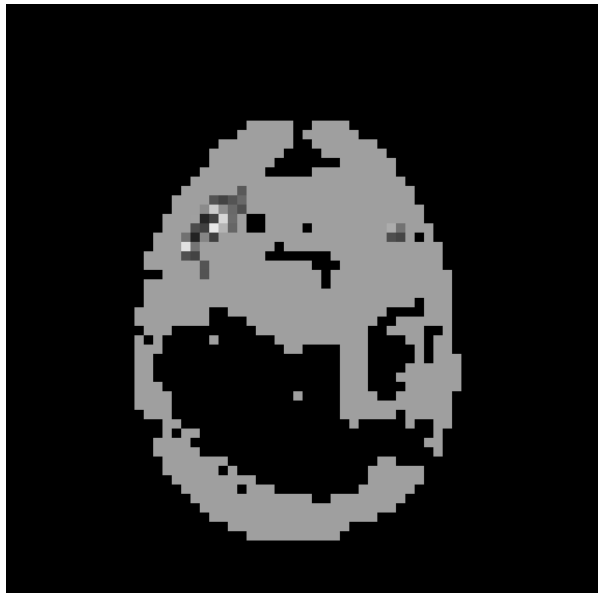
PGM: Converting to the ASCII PGM Format

So if we have a C++ program that can read ASCII PGM files, and we have grayscale data stored in a TIF file called *xray.tif*, we can use the **convert** program to make an ASCII PGM version of the same graphics information, called *xray.pgm*.

```
convert spine.tif -compress none spine.pgm
```

Specifying "-compress none" means we use the ASCII text format for the file, not the binary format.

PGM: Comparison of File Sizes



PGM: Comparison of File Sizes

	ASCII PGM	Binary PGM	PNG	JPG	TIFF
CASA	561,036	165,615	105,787	44,403	51,338
ROI_14	624,301	256,050	2,526	13,640	3,928
SURF	1,264,989	307,248	193,194	66,534	307,866

The ASCII PGM file can require 10 to 20 times more space than other formats, so it's not a good format for permanent storage of a lot of files. But making a temporary ASCII PGM copy of an image means your work of reading or writing the file will be easier.

PGM: Converting from the ASCII PGM Format

If we have a C++ program that creates an ASCII PGM file as output, we may need to use that graphics information with another program, which may require a different format.

Again, the **convert** program can help. When converting from the PGM format, there is no need to specify whether the PGM file used the ASCII or binary format, so the command is simple.

To create a JPG version of the PGM file *spine.pgm*, we would type

```
convert spine.pgm spine.jpg
```

Image Processing

- Introduction
- Working with Files
- Example Images
- The Portable Gray Map Format
- **The Noise Problem**
- The Edge Problem
- Lab Exercise #11

SALT: Extreme Noise

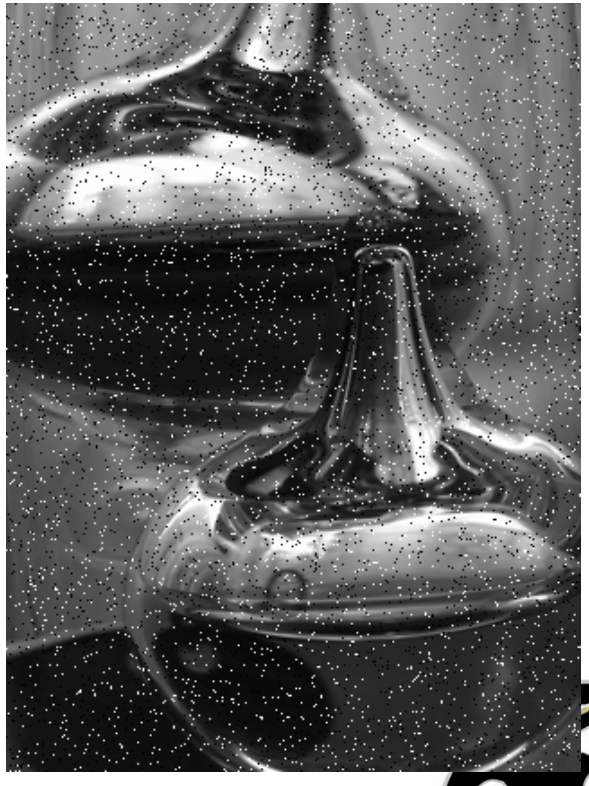
Sometimes an image can have a more serious problem than being dark or washed out. The physical process of recording and storing an image is subject to disturbance and damage.

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: RGB and Grayscale Examples



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. We might try to replace every pixel by the average of its neighbors; this would dilute the bad values, but they would still be visible. It would be better if we could make the bad values disappear.

SALT: The Median

The average is an attempt to produce one value that fairly represents all the values present, by summing the values and dividing by their number.

For our noisy problem, we expect cases where one value is essentially meaningless because it is extreme; 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. A single extreme value will generally have no effect on the result!

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

SALT: Computing The Median

One way to compute the median of N numbers is first to sort them, and then to choose the entry at index $\frac{n}{2}$.

```
int median ( int n, int p[] )
{
  int value;

  bubblesort ( n, p );

  value = p[n/2];

  return value;
}
```

SALT: Using the Median Filter

To apply the median filter to a noisy image stored in the 2D array **p[m][n]**, we make a new array **p2[m][n]**.

The color of the pixel at location p2[i][j] is determined as the median of the colors of the old pixels in the "neighborhood", that is, somewhat symbolically;

```
                                        p[i-1][j]
p2[i][j] = median ( p[i][j-1]     p[i  ][j]     p[i][j+1] )
                                        p[i+1][j]
```

SALT: Images After Median Filtering

If we repeated the filtering one more time on the color pictur almost all the remaining dots would disappear.

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 repeating the filtering operation, or by using a larger sample of pixels. However, this will mean that we gradually introduce some blurriness into the picture.

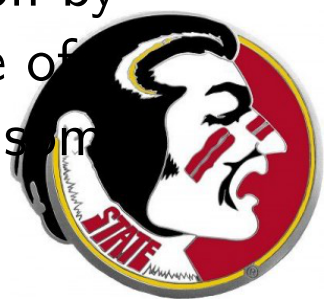


Image Processing

- Introduction
- Working with Files
- Example Images
- The Portable Gray Map Format
- The Noise Problem
- **The Edge Problem**
- Lab Exercise #11

EDGE: An Automatic Organization Tool

The eye is an automatic image processor. The eye is very good at detecting **edges**.

The eye and the brain use edges to organize the bits of light and color into a model of the physical world, with trees and tigers and friends.

We will try to understand how the eye can recognize an edge, by sketching out a computer program to find edges in an array of image information.

EDGE: What is an Edge?

An edge is a surprising (and possible dangerous!) event.

EDGE: What is an Edge?

One place we encounter an edge is when walking along a sidewalk. The sidewalk need not be flat; it might be sloping up a hill. However, even if we close our eyes, we can follow the slope of the sidewalk, because it changes in a gradual and regular way.

If we suddenly step off a curb, we are shocked. Our mental model of the sidewalk has "broken". It seems as though an edge represents a difference between what we expect and what we encounter. Something new is about to occur!

If we think of this mathematically, a smoothly changing sidewalk is like a a function that is linear, or perhaps has a derivative that changes, but slowly and smoothly.

Perhaps an edge is a place where the derivative is large.

EDGE: Differences Measure Change

Our image is not a function, so we can't compute its derivative.

However, the derivative is intended to describe changes per unit step. Since our data is stored in a grid, we can think of how the data changes going one step left or right, up or down. These measures of change are similar to a derivative.

If our image data is stored in a pixel array called P[][], we could estimate the "right/left" and "top/bottom" changes at P[I][J]:

```
left to right change =  P[i  ][j+1] - P[i  ][j-1]
bottom to top change =  P[i+1][j  ] - P[i-1][j  ]
```

```
              P[i+1][j]
P[i][j-1]     P[i  ][j]   P[i][j+1]
              P[i-1][j]
```

EDGE: Our Edge Detector

A large change, in either direction, says our image information is changing fast over a short range. This sounds like what we mean by an edge.

The sum of the absolute value of the differences is a measure of how fast things change at each pixel, which we'll call **E[i][j]**:

```
E[i][j] = | P[i][j+1] - P[i][j-1] |
        + | P[i+1][j] - P[i-1][j] |
```

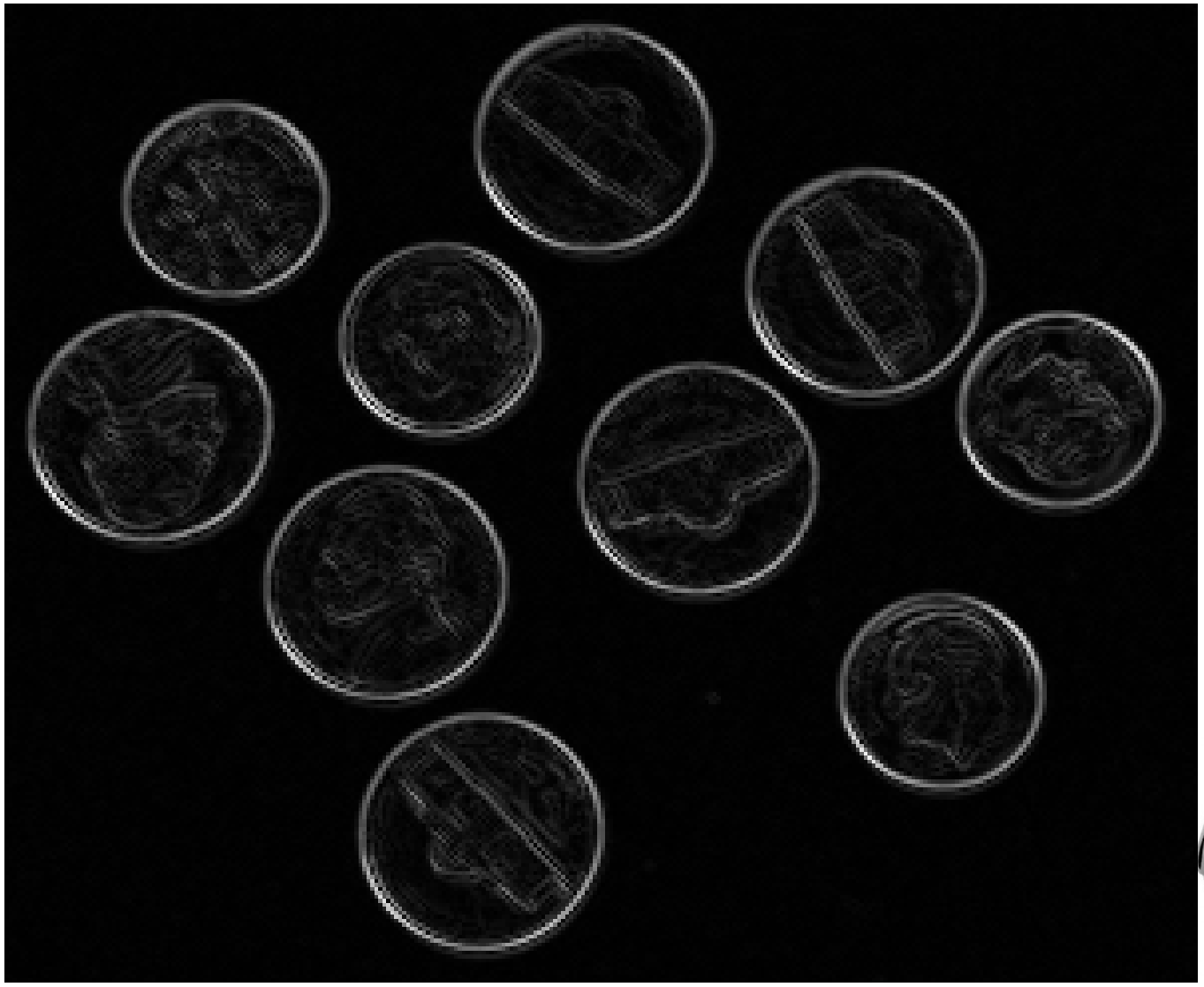
The value of **E** is zero at places where the pixels are "flat" and is large when nearby values differ a lot.

EDGE: A Test Image With Edges

EDGE: Compute E, and Normalize

```
//
//  Compute E[i][j] except for first and last row and column.
//
  emax = 0.0;
  for ( j = 1; j < n - 1; j++ )
  {
    for ( i = 1; i < m - 1; i++ )
    {
      e[i][j] = abs ( p[i+1][j] - p[i-1][j] )
              + abs ( p[i][j+1] - p[i][j-1] );
      if ( emax < e[i][j] )
      {
        emax = e[i][j];
      }
    }
  }
//
//  Normalize E[i][j] so the maximum value is 255.
//
  for ( j = 1; j < n - 1; j++ )
  {
    for ( i = 1; i < m - 1; i++ )
    {
      e[i][j] = e[i][j] * 255 / emax;
    }
  }
```

EDGE: The Value of E

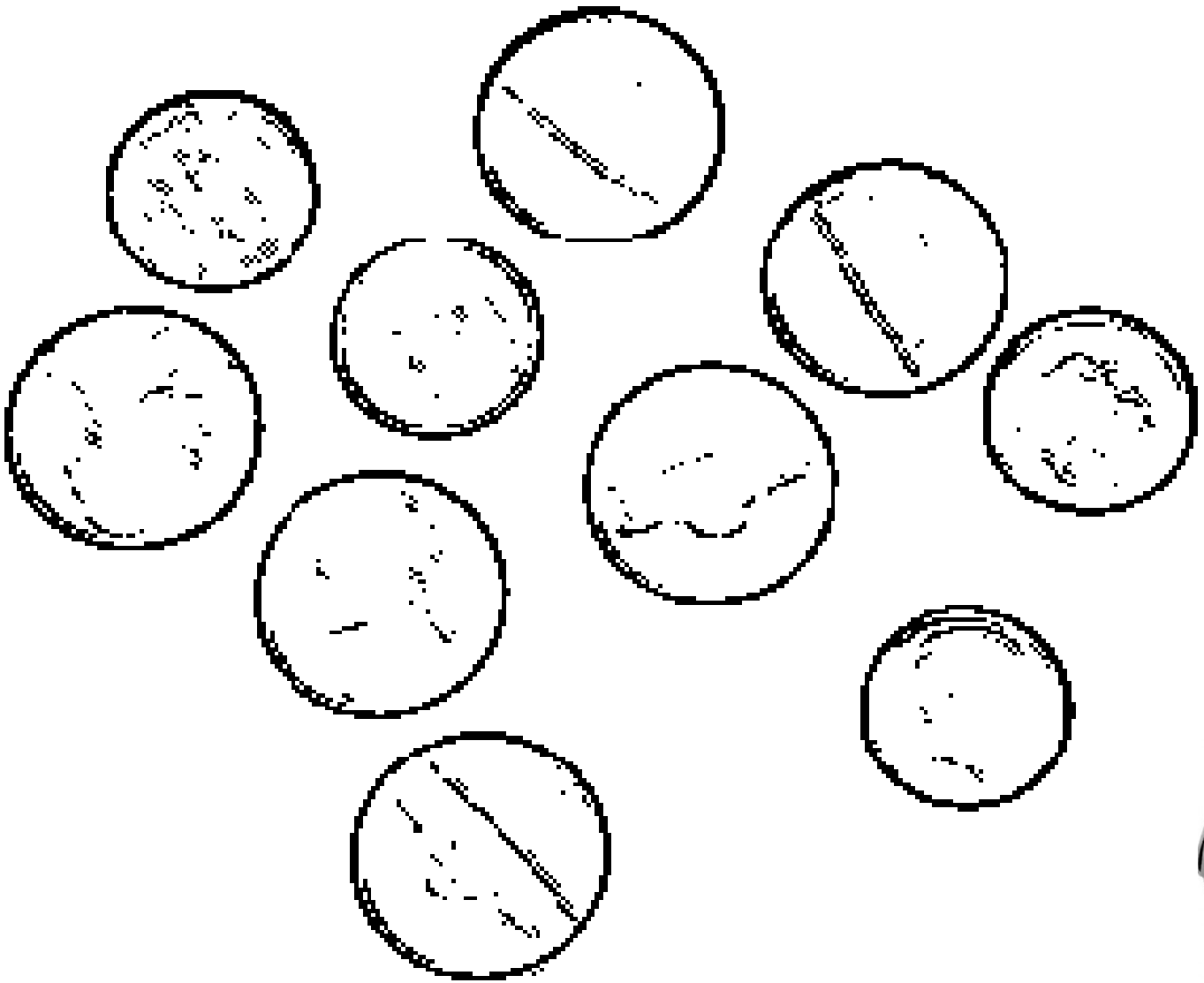


EDGE: Use a Threshold, and Reverse Video

Our image is showing shades of gray, but we need to decide what is an edge and what isn't. We can use a threshold that says that all values of **E** above a certain limit will be marked as edge pixels. This will have the effect of making all the pixels black (no edge) or white (edge) and no shades in between.

In fact, it's easier to see black lines on a white background than the other way around, so we'll follow the threshold operation by reversing the video.

EDGE: E with Threshold and Reverse Video



EDGE: More Is Needed!

Our computation and display of the quantity **E[][]** suggests how a computer could recognize edges in image data.

But there is still more work to do! Although we see that the coins have been identified, the fact that the black pixels actually form boundaries of coins is still something our eye is doing for us, not the computer!

So the computer would have to find a black pixel, look nearby for other black pixels, and join them into a continuous border for a coin. From that, we might be able to estimate the size, and hence the value, of the coin.

This simple problem, which our eyes "solve" every second, is cle pretty involved. And yet there are computer programs availab which can go through all these steps, and more, so that they ca identify a person from a database of facial images.

Image Processing

- Introduction
- Working with Files
- Example Images
- The Portable Gray Map Format
- The Noise Problem
- The Edge Problem
- **Lab Exercise #11**

EXERCISE: Anonymous Course Evaluation

Evaluation:

- Go to http://campus.fsu.edu/esussai and log in. The evaluation is anonymous.

Let Detelina know that you've completed the evaluation so you can get credit for the exercise!