Linear Pixel Shuffling and Subpattern Searching

Adam Stein

Abstract: **In this project, an investigation of Linear Pixel Shuffling for subpattern searching was performed using software created for this purpose. Linear Pixel Shuffling is a method to get a random-like permutation of pixels. Because the algorithm is generic, it has been used for displaying images, graphics rendering, image compression, image morphology, and digital halftones. The algorithm was compared against Space Domain Correlation, Frequency Domain Correlation, and Random Ordering for its quickness in matching a pattern. On average, Linear Pixel Shuffling found a partial match in a third of the time compared to Space Domain, Frequency Domain, and Random Ordering.**

1.0 Introduction

Traditional textbook subpattern searching algorithms are based on Space Domain Correlation and Frequency Domain Correlation. Both are mathematically equivalent, the only difference is the domain in which they operate. Other techniques, such as Random Ordering (picking the pixel locations at random), can be used in place of Space Domain and Frequency Domain. In this paper, Linear Pixel Shuffling (LPS) will be investigated as an additional methodology used for subpattern searching.

1.1 Space Domain

For every pixel location in the image, find the cross-correlation to the pattern at that location by summing up the pixel-by-pixel multiplications. This is done starting at location [0,0] and moving one pixel at a time across each scanline (i.e. [0,0], then [0,1]...[0,N], then [2,0]...[2,N] and so on until [N,N]).

1.2 Frequency Domain

Instead of the correlation being computed in the space domain, it is computed in the frequency domain. This is done by taking the Fourier Transform of image and pattern, multiplying the transforms together, and then taking the inverse Fourier Transform. There will be a peak in this field corresponding to the location of the pattern in the image. In mathematical theory, this gives you the same results as Space Domain Correlation.

1.3 Random Ordering

Random Ordering is a method to get a completely random ordering of pixels.

1.4 Linear Pixel Shuffling

LPS is a method to get a random-like permutation of pixels. Unlike other techniques, LPS creates a pixel ordering that is spread evenly across the image. It wraps around the image by use of modular arithmetic. A simple arithmetic progression is needed to determine the next pixel location. Refer to "Linear Pixel Shuffling for Image Processing, an Introduction”[1] and "Linear Pixel Shuffling Made Easy" by Peter Anderson (pga@cs.rit.edu) for more information.

1.5 Experiment Rational

Experiments were designed to test the speed and efficiency of LPS compared to three other algorithms: Space Domain, Frequency Domain, and Random Ordering.

The reason for comparing LPS against Space Domain is that Space Domain **is** the traditional textbook approach to pattern searching. It was hypothesized that, because LPS jumped around, it could find a pattern faster than Space Domain. It would take about the same amount of time for both algorithms to cover *every* pixel in an image, however with LPS, the pattern’s neighborhood could be found before each pixel location was searched. Even if LPS doesn’t find the exact center of the pattern before Space Domain, LPS would give a faster indication of where the pattern could be found within the image.

It’s well known that Frequency Domain can be much faster than Space Domain; the speed up factor depends upon the ratio between image and subpattern size. For large ratios, i.e. a 3x3 correlation for a 500x500 image, it is faster to work in the space domain. In case the subpattern size is a large fraction of the whole image size, it is faster to work in the frequency domain). This increase in speed is based upon the number of operations (adding, multiplying, etc.) that need to be done. In this experiment, the Frequency Domain algorithm was used to compare against LPS for cases where the Frequency Domain was faster than Space Domain.

Random Ordering has a large chance of being faster than Space Domain or Frequency Domain. The ordering of pixels is guaranteed that every pixel location will be investigated.

1.6 Hypothesis

The hypothesis of the present paper was that LPS could give faster approximate results. Instead of being sequential, LPS wraps around the image by use of modular arithmetic. LPS tries to space the jumps evenly. All that is needed to determine the next pixel in

1. P. G. Anderson, “Linear pixel shuffling for image processing: an introduction,” *Journal of Electronic Imaging*, vol. 2, no. 2, April 1993, pp. 147-154

sequence is a simple arithmetic progression. Even though the exact center of a pattern within an image might be unknown, LPS could indicate where that pattern could be found faster than other subpattern searching algorithms would. This was found to be, indeed, the case.

2.0 Objective

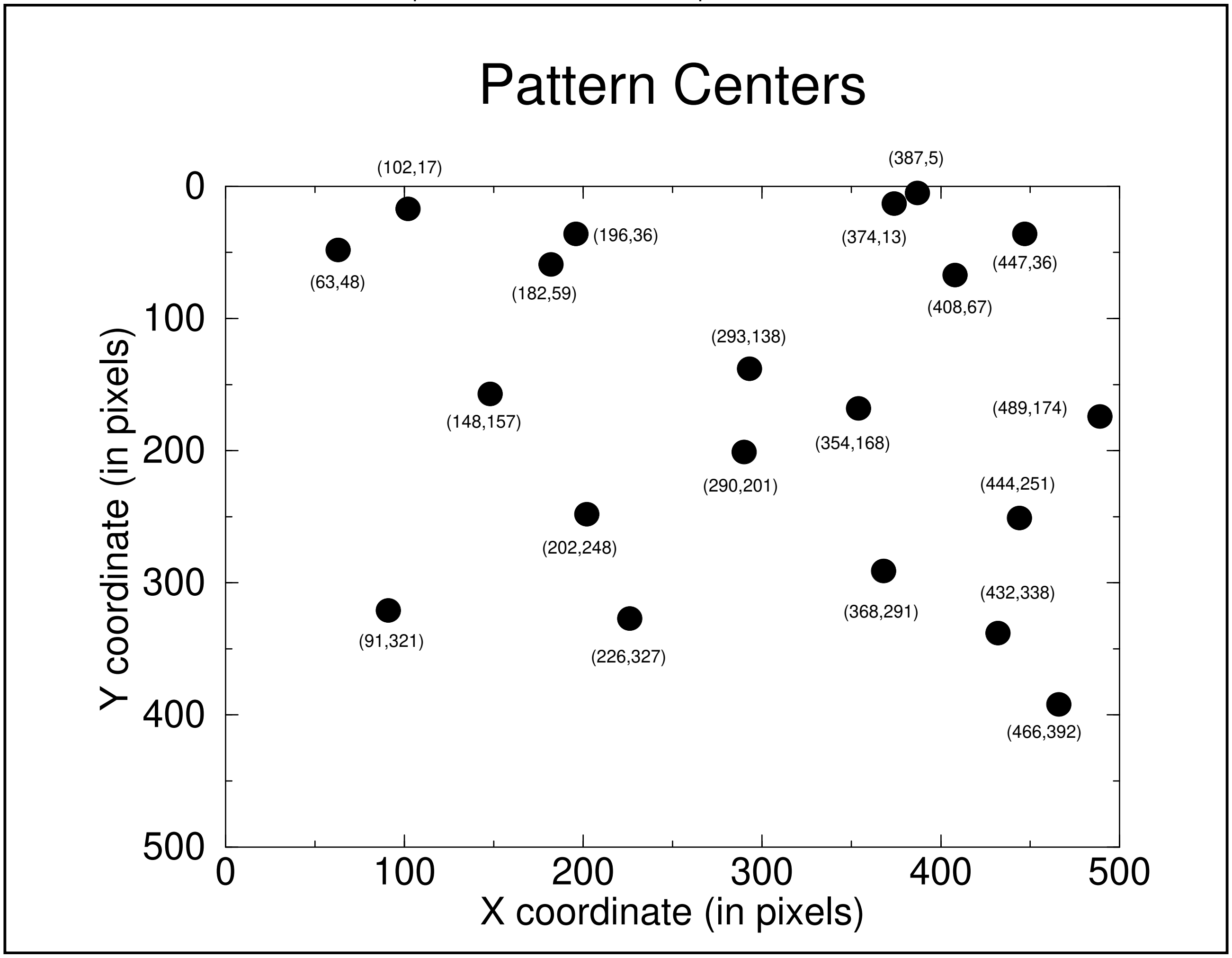
The objective of this project is to compare Linear Pixel Shuffling as a Subpattern Searching tool, against more traditional algorithms that are based on Space and Frequency Domain Correlation. The study will demonstrate the speed and efficiency of LPS as a potentially enhanced method for Subpattern Searching.

3.0 Experiment Design

There were three experiments. The first experiment kept the center of the pattern constant while changing the size. The second experiment kept the pattern size the same and moved the pattern around the image. The third experiment is a real world example.

Experiment 1 had five cases. The pattern sizes were: 3x3, 5x5, 11x11, 41x41, and 81x81 (all values are in pixel width and height). The image size was 500x500 pixels and consisted of a black rectangle (the pattern) on a white background. This first experiment was to see if LPS had any potential at all.

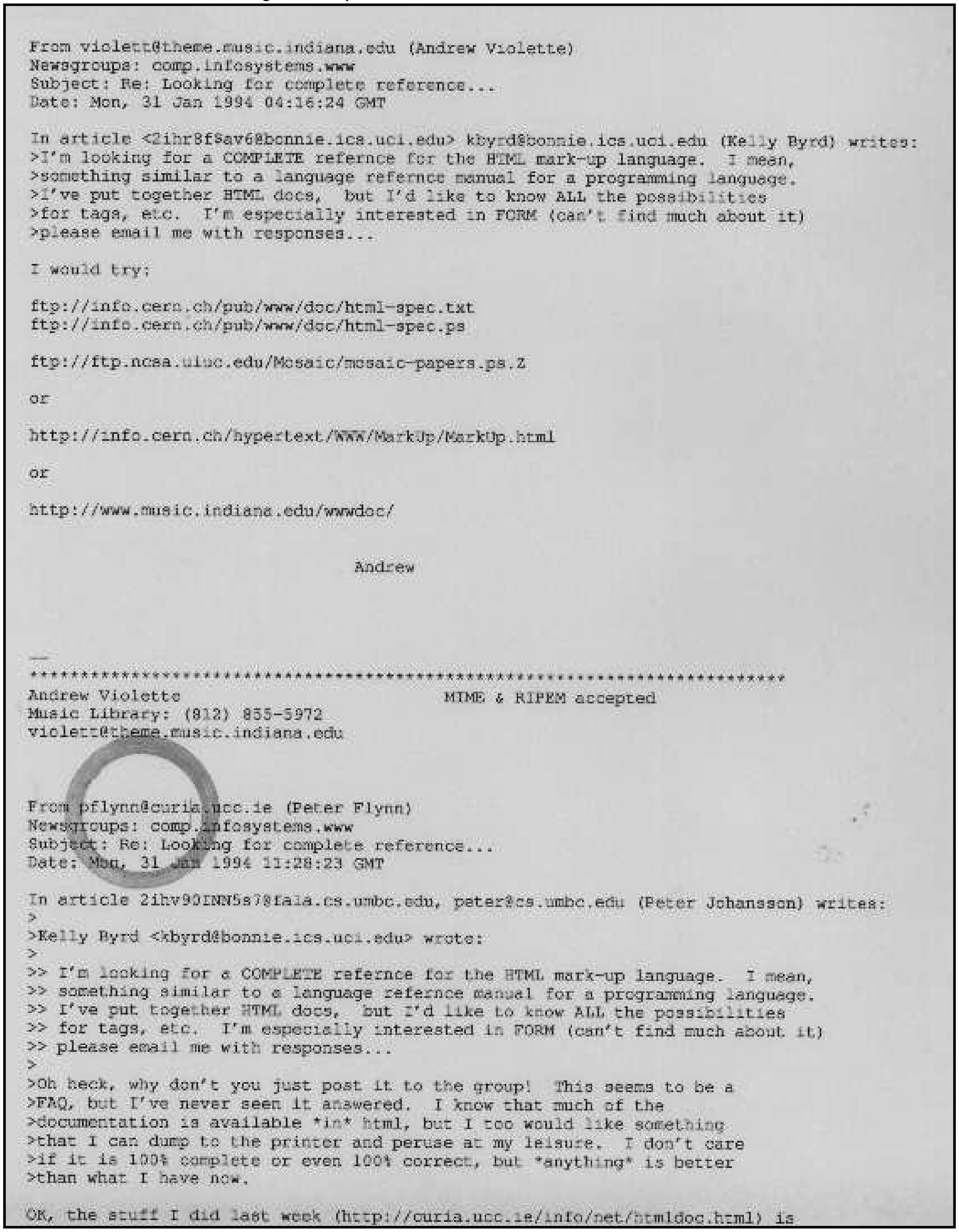
FIGURE 1. center positions for test cases in experiment 2



Experiment 2 had 20 cases. The pattern was chosen to be an 11x11 pixel square. The pattern size was chosen to be this large to give Random Ordering a chance. The center positions of the pattern can be seen in Figure 1 on page 4. The image size was 500x500 pixels and consisted of a black rectangle on a white background. This is to see if LPS is shift invariant or just lucky in Experiment 1 by jumping to the position in the image where the pattern was early in the execution.

Experiment 3 had one case (see Figure 2 on page 5). Unlike the first two experiments (in which the input data was computer generated with no noise), this experiment features a scanned page of text. The pattern to find is a circle on the text document. The circle was created by a rubber stamp. The image size was 612x792 pixels.

FIGURE 2. image for Experiment 3



From violett@theme.music.indiana.edu (Andrew Violette)
Newsgroups: comp.infosystems.www
Subject: Re: Looking for complete reference...
Date: Mon, 31 Jan 1994 04:16:24 GMT

In article <2ihr8f9av6@bonnie.ics.uci.edu> kbyrd@bonnie.ics.uci.edu (Kelly Byrd) writes:
>I'm looking for a COMPLETE refernce for the HTML mark-up language. I mean,
>something similar to a language refernce manual for a programming language.
>I've put together HTML docs, but I'd like to know ALL the possibilities
>for tags, etc. I'm especially interested in FORM (can't find much about it)
>please email me with responses...

I would try:

ftp://info.cern.ch/pub/www/doc/html-spec.txt
ftp://info.cern.ch/pub/www/doc/html-spec.ps

ftp://ftp.ncsa.uiuc.edu/Mosaic/mosaic-papers.ps.Z

or

http://info.cern.ch/hypertext/WWW/MarkUp/MarkUp.html

or

http://www.music.indiana.edu/wwwdoc/

Andrew

--
****
Andrew Violette MIME & RIPEM accepted
Music Library: (812) 855-5972
violett@theme.music.indiana.edu

From pflynn@curia.ucc.ie (Peter Flynn)
Newsgroups: comp.infosystems.www
Subject: Re: Looking for complete reference...
Date: Mon, 31 Jan 1994 11:28:23 GMT

In article 2ihv90INN5s7@fala.cs.umbc.edu, peter@cs.umbc.edu (Peter Johansson) writes:
>
>Kelly Byrd <kbyrd@bonnie.ics.uci.edu> wrote:
>
>> I'm looking for a COMPLETE refernce for the HTML mark-up language. I mean,
>> something similar to a language refernce manual for a programming language.
>> I've put together HTML docs, but I'd like to know ALL the possibilities
>> for tags, etc. I'm especially interested in FORM (can't find much about it)
>> please email me with responses...
>
>Oh heck, why don't you just post it to the group! This seems to be a
>FAQ, but I've never seen it answered. I know that much of the
>documentation is available *in* html, but I too would like something
>that I can dump to the printer and peruse at my leisure. I don't care
>if it is 100% complete or even 100% correct, but *anything* is better
>than what I have now.

OK, the stuff I did last week (http://curia.ucc.ie/info/net/htmldoc.html) is

For each experiment, each case ran using all four algorithms (Space Domain, Frequency Domain, LPS, and Random Ordering). The results were then plotted to compare distances (the Euclidean distance between where the object really was and where the procedure thought it was) and correlation values.

The distance values were plotted to see which algorithm found the pattern (or at least the area where the pattern was) the fastest.

Correlation values were plotted to see the semi-random progression of LPS compared to the other algorithms. In this type of graph, Space Domain and Frequency Domain would plot delta functions.

4.0 Experiment Implementation

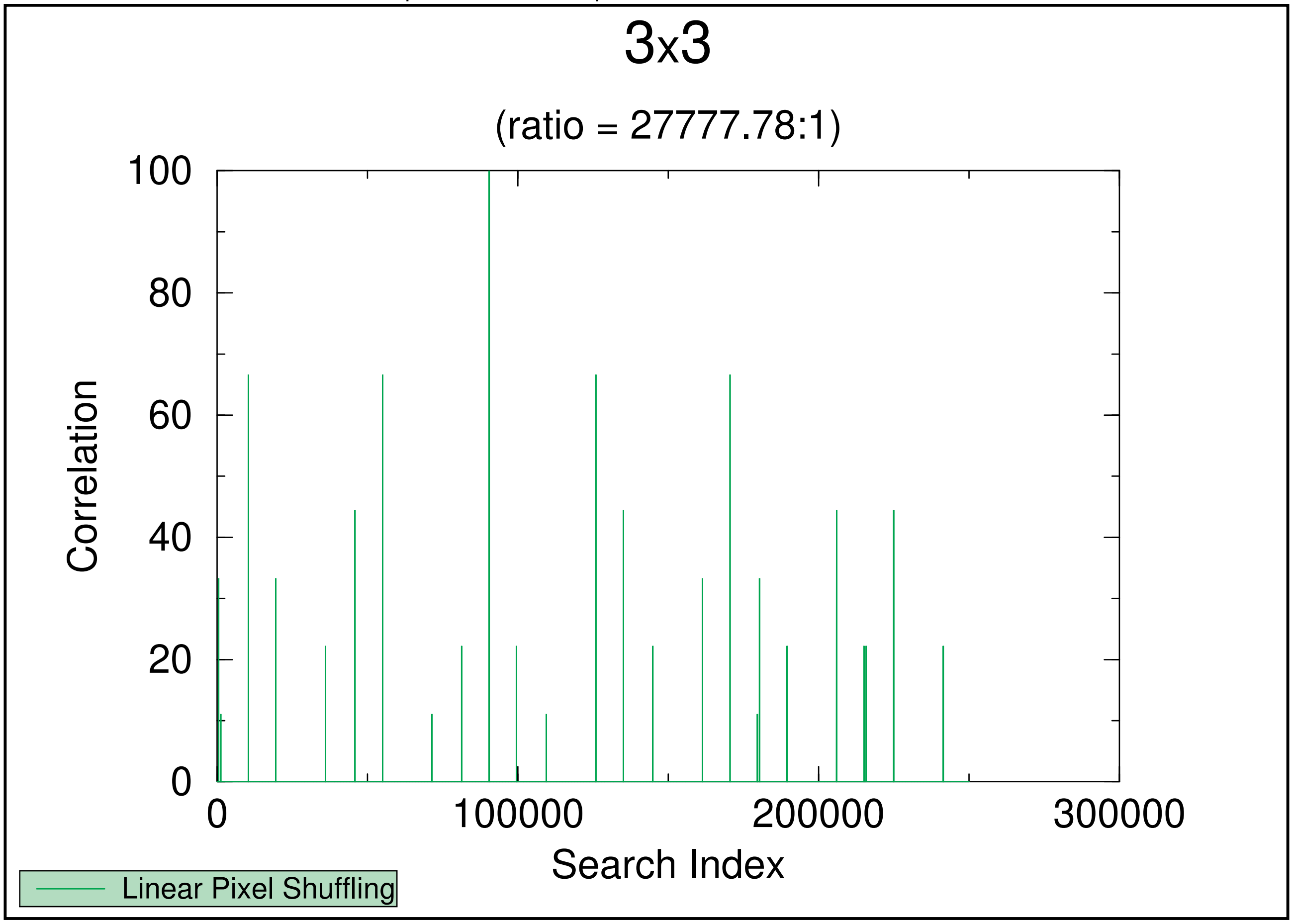
4.1 Sherlock

Software was created to compare the various algorithms. Sherlock is an Open Look™ application that can read in an image and pattern and run the four algorithms. The image and pattern files are in an internal format which consists of a simple header followed by bytes of data (1 byte/pixel). Currently, the image and pattern must be grayscale (0 = black, 255 = white).

Sherlock was written in C using the XView toolkit on a Sun SPARCStation 4/40. Within Sherlock, two external routines were imported: a Fast Fourier Transform routine originally written in Fortran by Norman Brenner in 1968, which was translated to C by Keith Knox of Xerox Webster Research Center in September 1990; and timing macros by George V. Neville-Neil, Universiteit Twente (gnn@cs.utwent.nl). The software can run either in GUI mode or in batch mode. Batch mode can process commands from a file or interactively from the keyboard. Two result files are created for each algorithm run, the correlation results, and the distance results. An example plot of correlation values for LPS can be seen in Figure 3 on page 7. An example distance plot for all four algorithms can be seen in Figure 4 on page 8.

The correlation values are the amount of correlation between image and pattern at each pixel location within the image. These are not **true** correlation values in the sense of the Space Domain equation. In order to speed up processing, it was faster to find the difference (rather than the correlation) between image and pattern and then invert so that the maximum value showed the smallest difference. So, in effect, these values are really differences rather than correlations. The search index is the pixel ordering. A search index of 100 is the 100th pixel coordinate correlated. For Space Domain, this graph would show a delta function.

FIGURE 3. sample correlation output



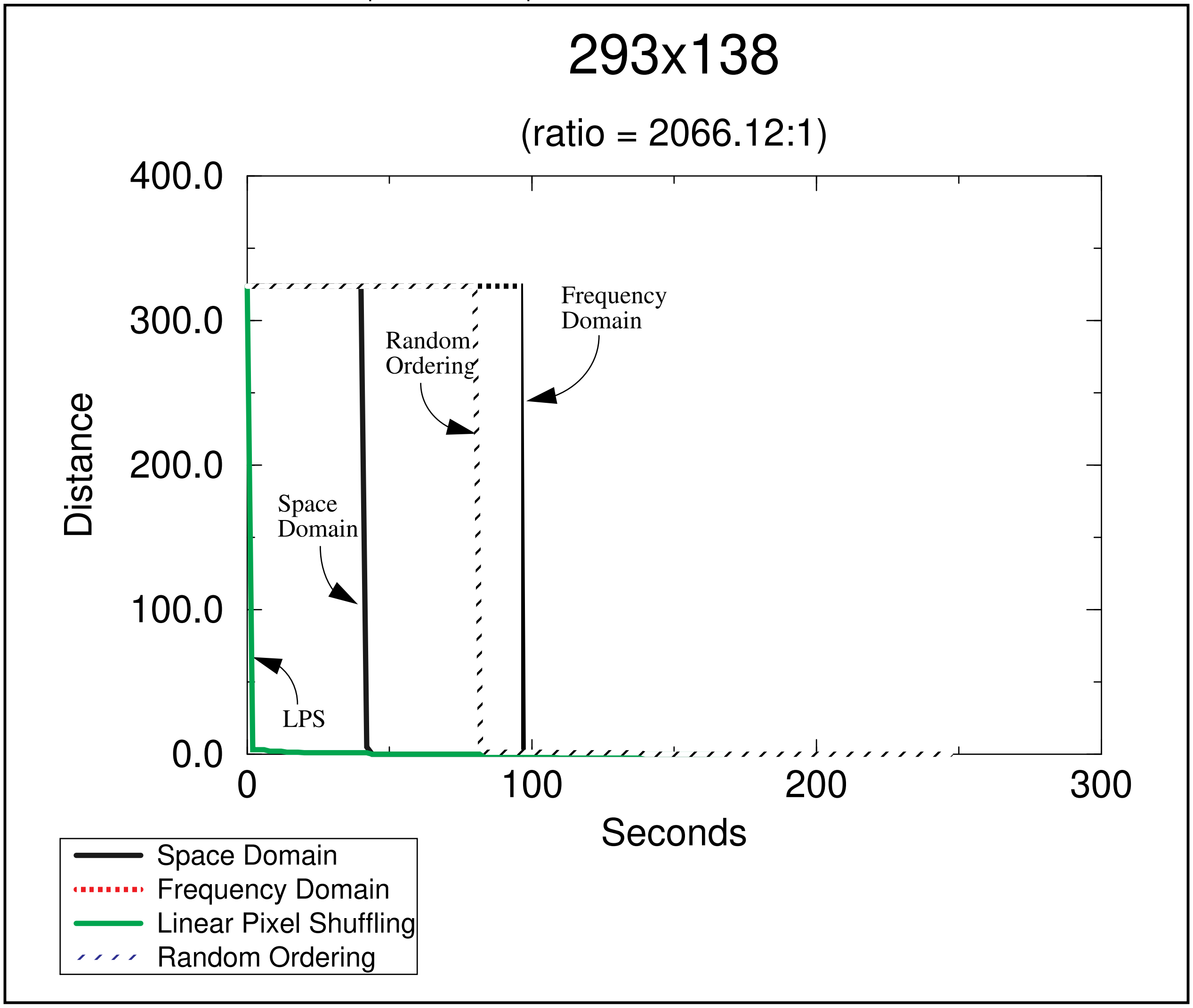
The other results file is comprised of the Euclidean distance between where Sherlock thinks the pattern is and where it really is. Distance is mathematically defined as the distance between 2 points given by:

(EQ 1)

$$D = \sqrt{(x_1 - x_0)^2 + (y_1 - y_0)^2}$$

where (x_0,y_0) are the true coordinates of the pattern and (x_1,y_1) are the estimated coordinates.

FIGURE 4. sample distance output



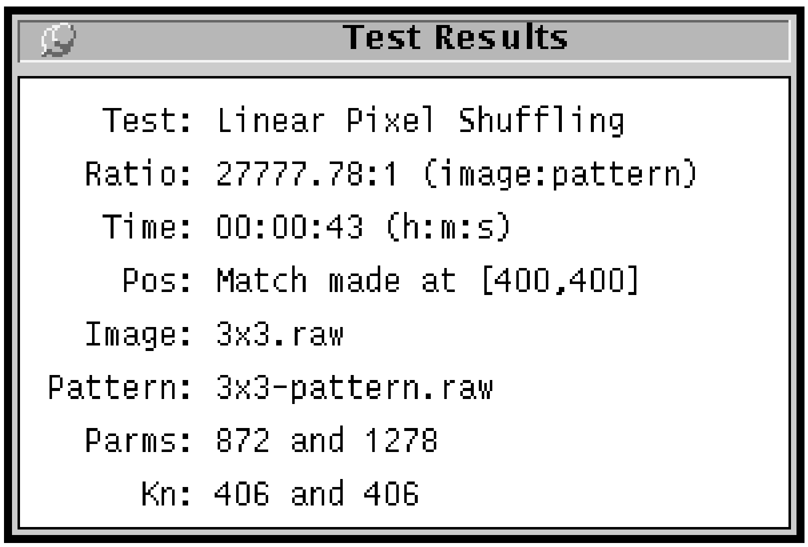
When an algorithm starts, the estimated position of the pattern is set to (0,0). This means that if the pattern is near location (0,0), the distance will start off small as opposed to a pattern that is further away.

The results files are formatted to be read in with *ACE/gr*, a free 2D graphing program for exploratory data analysis (xvgr - Open Look™ or xmgr - Motif version) written by Paul Turner (of the Oregon Graduate Institute, email pturner@ese.ogi.edu). This enables the results to be graphed quickly.

After an algorithm has run, information about it is presented. In batch mode, it is written to the screen. In UI mode, a window pops up with the information (see Figure 5 on page 9). The information given is the same in either case. This includes:

Test - algorithm that ran
Ratio - ratio between image and pattern (image:pattern)
Time - time spent in running the algorithm (h:m:s)
Pos - position of where a match was made
Image - filename of the image used
Pattern - filename of the pattern used (or *from image* if cropped)

FIGURE 5. Sherlock results



In the case of LPS, the parameters (Parms) and the increments (Kn) used are given. In the case of Random Ordering, the seed (Seed) for the random number generator is given.

4.1.1 Command Line Options

-i filename
: is the name of the **image file** to use. This option is required. This file must be in *Sherlock* internal format (see the header_util utility).

-b { batchfile | - }
: specifies to run in **batchmode**. No graphical UI is shown in this mode. A file of commands (**batchfile**) can be given to be interpreted. If a "-" is given for a filename, batch mode commands are read in from **stdin**. See Section 4.1.3, “Batch Mode,” on page 13. A graphical UI will be presented if this option isn't given.

-ii { y | n }
: Determine how to invert the **image** for the Frequency Domain Correlation algorithm. See Section 4.3, “Frequency Domain,” on page 15. A "y" will

always invert and a "n" will never invert. By default, *Sherlock* will automatically determine if inversion is needed.

-ip { y | n }
: Determine how to invert the **pattern** for the Frequency Domain Correlation algorithm. See Section 4.3, "Frequency Domain," on page 15. A "y" will always invert and a "n" will never invert. By default, *Sherlock* will automatically determine if inversion is needed.

"-l 'p1 p2 xinc yinc'
: are the **parameters** to use for the Linear Pixel Shuffling algorithm. "p1" is the first maximum number. "p2" is the second maximum number. "xinc" is the X increment. "yinc" is the Y increment. All numbers are integers. See Section 4.5, "Linear Pixel Shuffling," on page 16 for more details on what these numbers should be and why they are needed. If not given, they are calculated using a modified Fibonacci sequence.

-p filename
: is the name of the **pattern file** to use. This file must be in *Sherlock* internal format (see the header_util utility).

-rc filename
: is the name to save the **correlation data** to. This file is written out to a format to be read in by **ACE/gr**. Default is "corr_results.data".

-rd filename
: is the name to save the **distance data** to. This file is written out to a format to be read in by **ACE/gr**. Default is "dist_results.data".

-rs number
: is the **random seed** to use in the Random Ordering algorithm. Default is to use the number of seconds since 00:00:00 GMT, Jan. 1, 1970 as returned by the time(3) function.

-s subtitle
: is the **subtitle** for the output graphs (corr_results.data and dist_results.data). Default is to have no subtitle on the graphs.

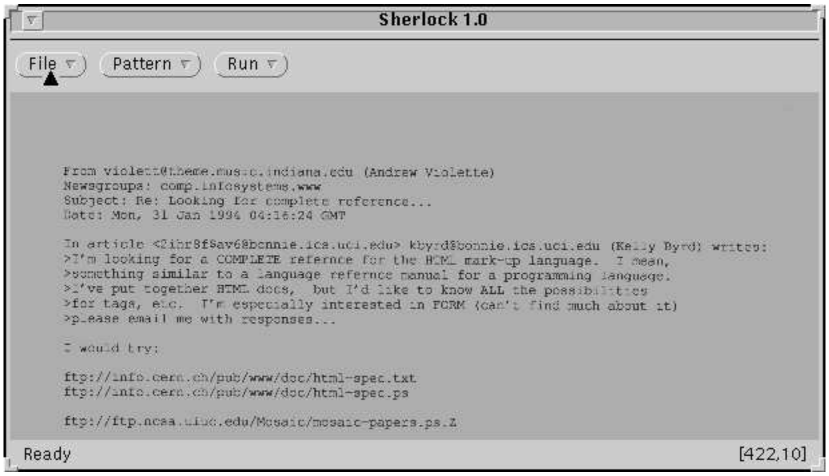
-t title
: is the **title** for the output graphs (corr_results.data and dist_results.data). Default is to have no title on the graphs.

-v displays the **version** number and quits. No further processing is done.

4.1.2 GUI Mode

By default, Sherlock will start in the interactive graphical UI mode. The image will be displayed with a command panel (of button menus) above it (see Figure 6 on page 11). If the pattern is specified on the command line, it is displayed in it's own window. The left footer of the image and pattern windows displays Sherlock's current state (i.e. Ready, Loading..., etc.).

FIGURE 6. main Sherlock window

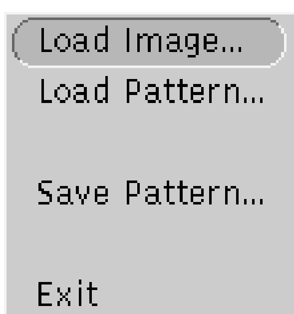


The button menus on the command panel are labeled **File**, **Pattern**, and **Run**.

The **File** button menu contains I/O options and "Exit". Within this menu, the following choices are available:

Load Image...	-	load a new image into memory
Load Pattern...	-	load a (new) pattern into memory
Save Pattern...	-	save a pattern to disk
Exit	-	exit Sherlock

FIGURE 7. Sherlock file menu



Selecting the load/save options will bring up a window asking for a directory and a filename. After this information is entered, the user can either click the "Apply" button to apply the input or "Cancel" button to discard the changes. Values entered for directory and filename will remain with the individual windows (i.e. a filename given to the Load Image window will appear the next time this window is displayed and won't appear for the other I/O windows).

The “Save Pattern...” choice is grayed out until there is a pattern in memory to save.

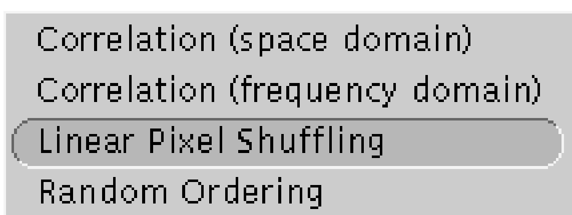
Files to be loaded must be in Sherlock internal format. Files are saved in this internal format.

The **Pattern** button menu contains choices for dealing with patterns. The “Select” choice will crop the pattern from the image. The mouse can be used to create a crop box (see Mouse Operations paragraph below). The “Display” choice will display a pattern that’s in memory. Both choices are grayed out until needed (for “Select” that’s creating a crop box, for “Display”, that’s having a pattern in memory to display).

FIGURE 8. Sherlock pattern menu

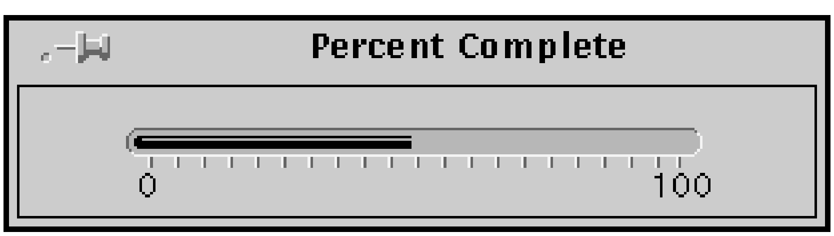
The **Run** button contains the algorithms to run. The four choices are: “Space Domain Correlation”, “Frequency Domain Correlation”, “Linear Pixel Shuffling”, and “Random Ordering”. This button is grayed out until there is a pattern in memory so that the algorithms can run.

FIGURE 9. Sherlock run menu



A gauge is displayed when an algorithm is being executed. This gauge displays how much of the algorithm run time has been spent and how much more there is to go.

FIGURE 10. Sherlock status gauge



Mouse Operations are as follows.The crop box used to select a pattern from the image is drawn by holding down the left button. The box can be stretched by moving the mouse with the left button held down. As soon as the left button is released, the crop box can no longer be adjusted. If the left button isn't being held down, the current mouse position is displayed in the right footer of the image window. This allows the user to see how good the algorithm worked to find the center location of the pattern.

The colormap is set to a grayscale colormap (and so the images are limited to 8 bit/pixel grayscale). Closest colors are used if a certain grayscale is unavailable (meaning that a white background can look slightly grayish). The data used in the calculations or saved to disk *are* the actual values, not the colormapped values.

4.1.3 Batch Mode

Sherlock can also run non-interactively in **batch mode**. In this mode, no images are displayed. This mode can be used in a pure ASCII environment.

A file of commands can be given to be interpreted. If a "-" (dash) is given as the filename for the "-b" option, commands are read from **stdin**.

The following commands are available:

crop x y width height
: Crop out an area from the image to become the pattern. **X** and **Y** are the (X,Y) starting coordinates of the area to crop. **Width** is the width (in pixels). **Height** is the height (in pixels). For example, the command:

    crop 10 20 100 150

  will crop an area from the image starting at pixel position (10,20) and crop out an area 100 pixels width by 150 pixels height. This cropped area becomes the pattern.

exit Exit the program.

invert type {yes|no|auto}
: Determine how to invert the image or pattern. This is the same as the "-ii" and "-ip" switches above (with the addition of "auto"). The **type** parameter determines which data is affected. It can be "image" or "pattern". Setting to "yes" will always invert. Setting to "no" will never invert. Setting to "auto" will let *Sherlock* try to automatically determine if inversion is needed. "Auto" is the default setting. An example:

    invert pattern no

  will cause *Sherlock* to not invert the pattern. This is useful if the pattern's (or image's) first pixel isn't the background color. See Section 4.3, "Frequency Domain," on page 15 for an explanation on why inversion is sometimes necessary.

load type filename
: Load a file into memory. The **type** parameter determines what is being read in (image or pattern). **Type** can be "image" or "pattern". The **filename** is the name of the file to read in. For example, the command:

    load image abc.raw

  will load the image in "abc.raw" into memory.

run algorithm
: Run the selected algorithm. The possible choices for **algorithm** are:

space	- space domain correlation
frequency (or freq)	- frequency domain correlation
lps	- linear pixel shuffling
random	- random ordering

For example, the command:

run lps

will run the Linear Pixel Shuffling algorithm.

save type filename

Save a file to disk. The **type** parameter determines what is being written out (image or pattern). **Type** can be "image" or "pattern". This version currently only supports saving patterns, so in effect **type** is useless at this point (i.e. the pattern will be saved regardless of what **type** is set to). The **filename** is the name of the file to read in. For example, the command:

save pattern abc.raw

will save the pattern from memory to "abc.raw".

seed = { number | ? }

Set the random seed to the number indicated. If the '?' is given instead of a number, *Sherlock* will get the random seed for the Random Ordering algorithm from the time(3) function. For example, the command:

seed = 471

will set the seed for the random number generator to 471.

sleep seconds

Sleep for the time indicated. The time argument (**seconds**) as you might have guessed are in seconds. For example, the command:

sleep 120

will sleep for 120 seconds (2 minutes).

!command

Execute a shell escape. **Command** is the command to run. For example, the command:

!ls -l

will run the "ls -l" command in the current directory.

4.2 Space Domain

Instead of the normal correlation of adding up the pixel-by-pixel multiplies, *Sherlock* sums the absolute value of the difference between corresponding pixels. This is given by:

(EQ 2)

$$g(\vec{q}) = \sum_{\vec{p} \in D} \left| f(\vec{p}) - h(\vec{p} + \vec{q}) \right|$$

where **f** is the pixel in subimage **D** and **h** is the image.

Sherlock then looks for a minimum to find the greatest correlation (i.e. the least difference between matrices). This speeds up the processing by not having to:

1. calculate the average for pattern and image
2. subtracting that average from each pixel

The average needed to be subtracted because good (and false) correlation results were coming from the totally white areas within the image.

Also, by not having to subtract the average, *Sherlock* can do the subtraction with integers, not floating point numbers.

The steps for this algorithm are as follows:

1. starting at [row=0,col=0], find the difference between pattern and image area
2. check difference value for minimum
3. move on to the next pixel position ([row,col+1] or if at the end of a scanline, [row+1,0])

4.3 Frequency Domain

The Fourier Transform function used assumes that the origin is in the middle of the scanline. However, *Sherlock* calculates using the origin as the first pixel in the scanline. In order to match up perfectly with the Space Domain Correlation, either "beercanning" or "checkerboarding" would have to be done so that the Fourier Transform function sees the data with the origin in the middle. “Beercanning” and “checkerboarding” are methods that can transform an image to move the origin from being the first pixel in a scanline to being in the center of a scanline (or visa versa). In order to save processing time, *Sherlock* doesn’t do either. It compensates by calculating the origin (middle) location of the pattern within the image by knowing that the Fourier Transform function used will always return the maximum correlation location at the upper left corner of the pattern. Had *Sherlock* done either "beercanning" or "checkerboarding", the maximum correlation would have been found in the center of the pattern, as it does for Space Domain.

In order for the Frequency Domain Correlation to work, *Sherlock* needs a black (pixel value 0) background with a non-black object. This is so zero padding of black can be applied to the pattern and image to boost the size to that of a power of 2 (so that the Fast Fourier Transform function can work as fast as possible). Therefore, the first pixel (location [0,0]) is checked to see if it’s equal to 0. If it is, *Sherlock* assumes the background is black and the object is non-black. If it isn’t, *Sherlock* inverts the image (or pattern). This assumes that the pattern won’t be covering position (0,0). If it is, or some reason, the auto detection isn’t working properly, the inversion behavior can be controlled by switches (-ii and -ip) and in the case of batch mode, by commands (invert type {yes|no|auto}).

The steps for this algorithm are as follows:

1. find power of 2 size equal to or greater than the largest dimension in pattern and image
2. zero pad pattern and image to that power of 2 size

3. calculate the two dimensional Fourier Transform of image and pattern
4. multiply transforms on a pixel-by-pixel basis
5. take inverse Fourier Transform of the product array
6. search for the maximum in the inverted transform results

4.4 Random Ordering

Random ordering is like SPACE DOMAIN (described above), except that selecting the next pixel within the image (step 3) is done by randomly choosing the next pixel location.

The pixel ordering is done so as to guarantee that every pixel location is visited only once.

4.5 Linear Pixel Shuffling

The change in position is calculated as:

New_X_coord = (Old_X_coord + X_increment) % X_max;

New_Y_coord = (Old_Y_coord + Y_increment) % Y_max;

The % symbol represents remaindering. Both the increments and the maximum values can be given on the command line (-l option). The maximum values (x_max, y_max) are equal to or greater than the image dimensions.

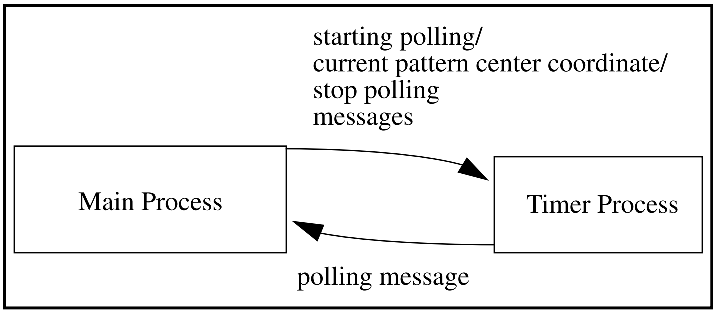
The parameters were obtained from a table of suitable parameters.

The algorithm steps are the same as for SPACE DOMAIN (see above) except for the method of selecting the next pixel within the image (step 3). Instead, the change in position is calculated using the equations listed above.

4.6 Methodology

To obtain the distance values, *snapshots* of where Sherlock thought the pattern was were taken at 2 second intervals during an algorithm's execution. This limits the time resolution so that the results can be ±1 second. This is done by a child timer process. When an algorithm starts executing, a message is sent to the timer process to begin polling every 2 seconds. At each interval, the timer process sends a message to the main Sherlock process asking for its guess of where the pattern center is. The main process responds with a message back to the timer process with the coordinates. The timer process then calculates the distance using these coordinates. When the algorithm has finished, the main process sends a message to the timer process telling it to stop polling.

FIGURE 11. block diagram of Sherlock and it's timer process



For the correlation data, the difference values (see Section 4.2 on page 14 which describes this) are saved during an algorithm's execution. After the algorithm is done, the following equation is used for Space Domain, LPS, and Random Ordering data to convert differences into normalized correlation values (from 0% to 100% correlation).

$$NC_i = \frac{\max D_i - D_i}{\max D_i} \times 100, (1 \le i \le P) \tag{EQ 3}$$

NC_i is the normalized output data correlation value. D_i is the difference value. P is the number of values. For Frequency Domain results, the data is normalized to compare to the other algorithm's results.

$$NC_i = \frac{C_i}{\max C_i} \times 100 \tag{EQ 4}$$

Since Frequency Domain calculates correlations (not differences), C_i is the correlation value.

5.0 Results

The results for each experiment are in two tables. The first table lists the number of seconds it took for each algorithm to find its first partial match of the pattern. The second table lists the number of seconds that it took to find the center of the pattern.

For the first two experiments, an algorithm was claimed to have found a partial match when the distance became smaller than the initial distance value. This is reasonable since there was no noise in the image and the higher correlation values could only have come from matching the pattern. The LPS parameters used for these experiments were 509 and 557 for the maximum values and 237 and 380 for the increments for X and Y, respectively.

Because the image in Experiment 3 had noise (and therefore no sharp indication of partial success in the correlation), it wasn't enough to say that the algorithm had found the pattern when the distance became shorter. Instead, the Space Domain results were used to find the maximum distance from first contact with the pattern to the center. This was known because Space Domain first makes contact with a pattern at its boundary and then moves toward the center (which results in a delta function). This maximum distance was used as a threshold. Any algorithm that had a distance equal to or smaller than this threshold value was considered to be matching the pattern and not something else. For this experiment, 619 and 809 were the maximum values and 288 and 552 were the increments for X and Y, respectively.

For all three experiments, the center of the pattern was considered matched when the distance was equal to 0.

5.1 Experiment 1

TABLE 1. first partial match found (in seconds)

Case	Space	Freq	Random	LPS
3x3	12	100	86	2
5x5	28	98	84	2
11x11	128	100	88	2
41x41	1,584	98	100	2
81x81	5,424	100	74	2

TABLE 2. found pattern center (in seconds)

Case	Space	Freq	Random	LPS
3x3	12	100	96	8
5x5	28	100	116	14
11x11	130	100	194	62
41x41	1,768	100	1,920	812
81x81	6,866	100	5,376	3,136

5.2 Experiment 2

TABLE 3. first partial match found (in seconds)

Case	Space	Freq	Random	LPS
(102,17)	4	96	84	2
(148,157)	48	98	76	2
(182,59)	16	98	90	2
(196,36)	8	98	84	2
(202,248)	76	96	86	2
(226,327)	102	96	86	2
(290,201)	60	98	82	2
(293,138)	42	98	82	2
(354,168)	50	98	92	2
(368,291)	90	98	88	2
(374,13)	2	96	76	2
(387,5)	2	98	78	2
(408,67)	18	140	72	2
(432,338)	104	96	88	2
(444,251)	76	96	88	2
(447,36)	8	98	88	2
(466,392)	124	98	78	2
(489,174)	52	96	74	2
(63,48)	12	98	92	2
(91,321)	100	98	106	2
Averages	49.7	99.4	84.5	2

TABLE 4. found pattern center (in seconds)

Case	Space	Freq	Random	LPS
(102,17)	6	96	146	44
(148,157)	50	98	126	22
(182,59)	20	98	148	158
(196,36)	12	98	210	112
(202,248)	78	96	200	86
(226,327)	106	98	88	168
(290,201)	64	98	196	62
(293,138)	44	98	140	44
(354,168)	54	98	216	92
(368,291)	92	98	178	98
(374,13)	4	96	104	156
(387,5)	2	98	234	38
(408,67)	22	140	164	28
(432,338)	108	98	120	68
(444,251)	80	98	106	100
(447,36)	12	98	160	8
(466,392)	128	98	210	106
(489,174)	56	96	192	154
(63,48)	16	98	176	62
(91,321)	104	98	124	104
Averages	52.9	99.7	161.9	85.5

5.3 Experiment 3

TABLE 5. first partial match found (in seconds)

Case	Space	Freq	Random	LPS
text	10,186	550	844	98

TABLE 6. found pattern center (in seconds)

Case	Space	Freq	Random	LPS
text	14,506	550	7,676	7,162

6.0 Analysis of Results

The following sections validate the results from Section 5.0 on page 17.

6.1 Experiment 1

First, let's look at Experiment 1. The following table shows how long it takes for pixels

TABLE 7. timing for Experiment 1 cases

Case	# pixels	Seconds/ Pixel	Pixels/ Second	# of pixels til 1st partial match	Seconds
3x3	200,400	5.9880E-05	16,700.0000	570	3.4132E-02
5x5	200,400	1.3972E-04	7,157.1429	570	7.9641E-02
11x11	200,400	6.4870E-04	1,541.5385	570	3.6976E-01
41x41	200,400	8.8224E-03	113.3484	5	4.4112E-02
81x81	200,400	3.4261E-02	29.1873	5	1.7131E-01

to be processed. The values were generated from the Space Domain. Since Space Domain processes sequentially in row order, we know how many pixels in the image the algorithm had to process before finding the center location of the pattern. Using this information, we can calculate the *seconds/pixel* (or *pixels/second*) rate. This rate can then tell us how long it took to go through N pixels.

For the table above, column 1 is the case (in terms of pattern size). Column 2 is the number of pixels to go through to get to the center of the pattern. Column 3 is the *seconds/pixel* rate based upon the number of pixels and the time it took to process them. For example, for the first case (3x3), we know from Table 2 on page 18 that it took 12 seconds for Space Domain to find the center. Since the center is at (400,400), we know column 2 to be 200,400 pixels. Just divide 12 by 200,400 to get the *seconds/pixel* rate. Column 4 is the inverse of column 3 to show the *pixels/second* rate. Column 5 is the number of pixels that had to be processed by LPS until a partial match was made. This is determined by looking at the pixel ordering that LPS went through. Since the same parameters were used for LPS in Experiments 1 and 2, this ordering will be constant. Column 6 is the number of seconds it took to get through the number of pixels listed in column 5.

Since each case has a different sized pattern, the time it takes to calculate the correlation at each pixel location in the image is different. The bigger the pattern, the more calculations are needed, the longer the time (as shown in columns 3 and 4). Column 6 is based on the individual times for each cases.

As can be seen in column 6, LPS finds a partial match in well under 2 seconds.

FIGURE 12. Pattern Size vs. Number of seconds to find a partial match in Experiment 1

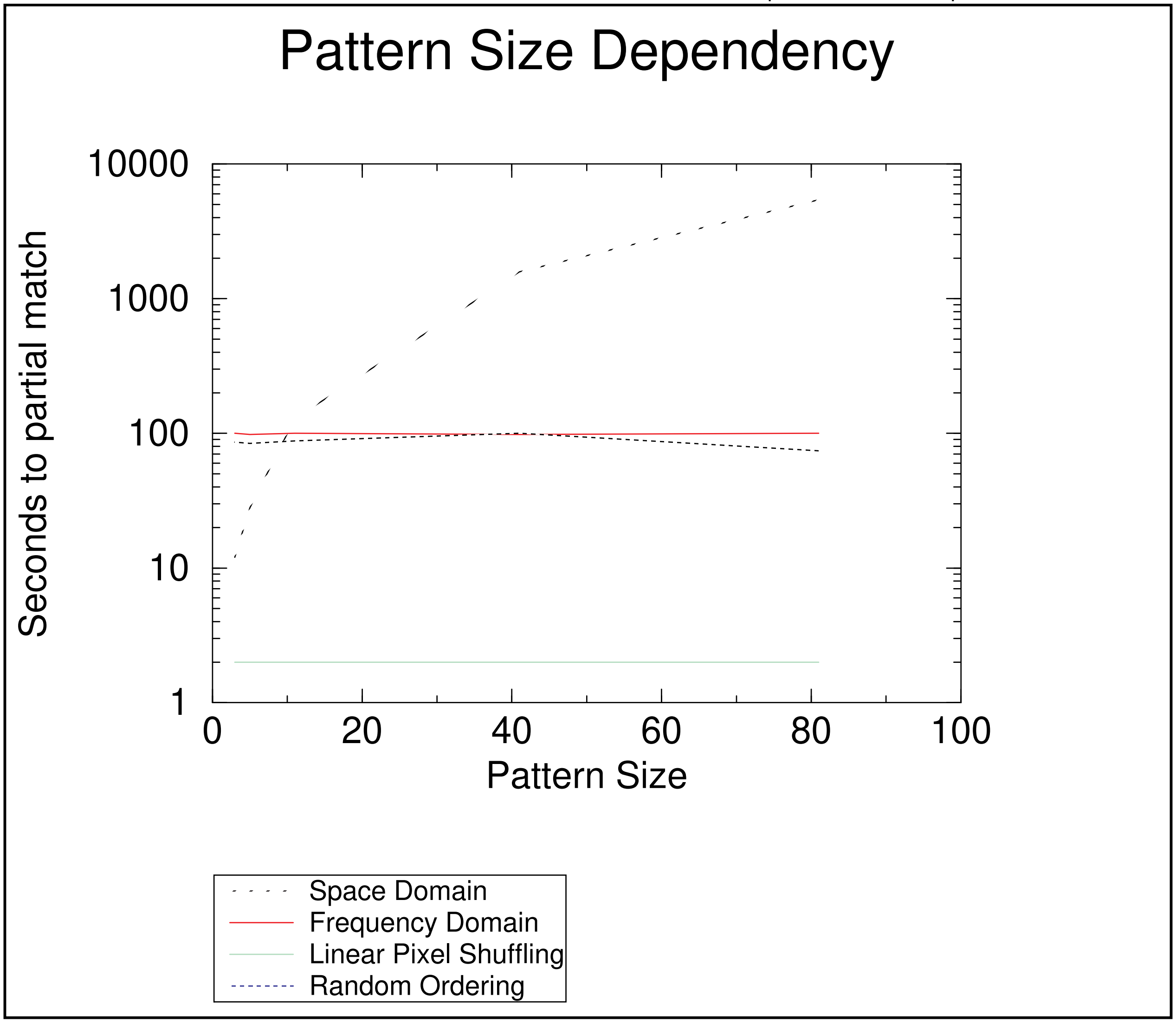


Figure 12 on page 22 shows the pattern size vs. the number of seconds it took an algorithm to find a partial match. This graph shows LPS, Frequency Domain, and Random Ordering to be pixel size invariant (with LPS being the quickest of the three) compared to Space Domain which is dependent on pattern size.

6.2 Experiment 2

TABLE 8. timing for Experiment 2 cases

Case	# pixels	Seconds/ Pixel	Pixels/ Second	# of pixels til 1st partial match	Seconds
(102,17)	8,602	6.9751E-04	1,433.6667	98	6.3374E-02
(148,157)	78,648	6.3574E-04	1,572.9600	583	3.7701E-01
(182,59)	29,682	6.7381E-04	1,484.1000	160	1.0347E-01
(196,36)	18,196	6.5949E-04	1,516.3333	4	2.5867E-03
(202,248)	124,202	6.2801E-04	1,592.3333	555	3.5891E-01
(226,327)	163,726	6.4742E-04	1,544.5849	412	2.6643E-01
(290,201)	100,790	6.3498E-04	1,574.8438	191	1.2352E-01
(293,138)	69,293	6.3498E-04	1,574.8409	525	3.3951E-01
(354,168)	84,354	6.4016E-04	1,562.1111	151	9.7648E-02
(368,291)	145,868	6.3071E-04	1,585.5217	279	1.8042E-01
(374,13)	6,874	5.8190E-04	1,718.5000	60	3.8801E-02
(387,5)	2,887	6.9276E-04	1,443.5000	416	2.6902E-01
(408,67)	33,908	6.4881E-04	1,541.2727	442	2.8583E-01
(432,338)	169,432	6.3742E-04	1,568.8148	338	2.1858E-01
(444,251)	125,944	6.3520E-04	1,574.3000	159	1.0282E-01
(447,36)	18,447	6.5051E-04	1,537.2500	43	2.7807E-02
(466,392)	196,466	6.5151E-04	1,534.8906	362	2.3410E-01
(489,174)	87,489	6.4008E-04	1,562.3036	604	3.9059E-01
(63,48)	24,063	6.6492E-04	1,503.9375	63	4.0741E-02
(91,321)	160,591	6.4761E-04	1,544.1442	638	4.1258E-01
Averages	82,473.10	6.4668E-04	1,548.5105	304.15	1.9669E-01

The descriptions for the columns are the same as for the table in the last section. Column 1 lists the cases in terms of pattern centers. Unlike Experiment 1, the pattern is the same size for all cases. This should lead us to believe the timings for columns 3 and 4 should be the same. They do differ by a small amount. The mean for *seconds/pixel* is **6.4668E-04**. This is roughly **26** times greater than the standard deviation (**2.4581E-05**). Column 6's results are based on this mean value.

Again, it can be seen that LPS finds a partial match under 2 seconds.

The results for Frequency Domain in the 408x67 case obviously don't seem to belong. This anomaly is explained by background processes running at the same time. Repeat trials of this case show the results to be similar to other case results.

TABLE 9. results for 5 retrials of case 408x67 for partial match (in seconds)

Case	Space	Freq	Random	LPS
retrial 1	20	99	42	2
retrial 2	18	99	39	2
retrial 3	18	98	114	2
retrial 4	18	98	82	2
retrial 5	20	99	80	2
means	18.8	98.6	71.4	2

TABLE 10. results for 5 retrials of case 408x67 for pattern center (in seconds)

Case	Space	Freq	Random	LPS
retrial 1	24	99	120	28
retrial 2	22	99	158	28
retrial 3	22	98	268	26
retrial 4	22	98	176	26
retrial 5	22	99	100	28
means	22.4	98.6	164.4	27.2

The same approximate values resulted for Space Domain, LPS, and Random Order as in the original test case. Rerunning this particular case shows that the 140 seconds for Frequency Domain from the original experiment is not correct. The mean values for Frequency Domain are close to other case results, as expected.

FIGURE 13. Position vs. Number of seconds to find a partial match in Experiment 2

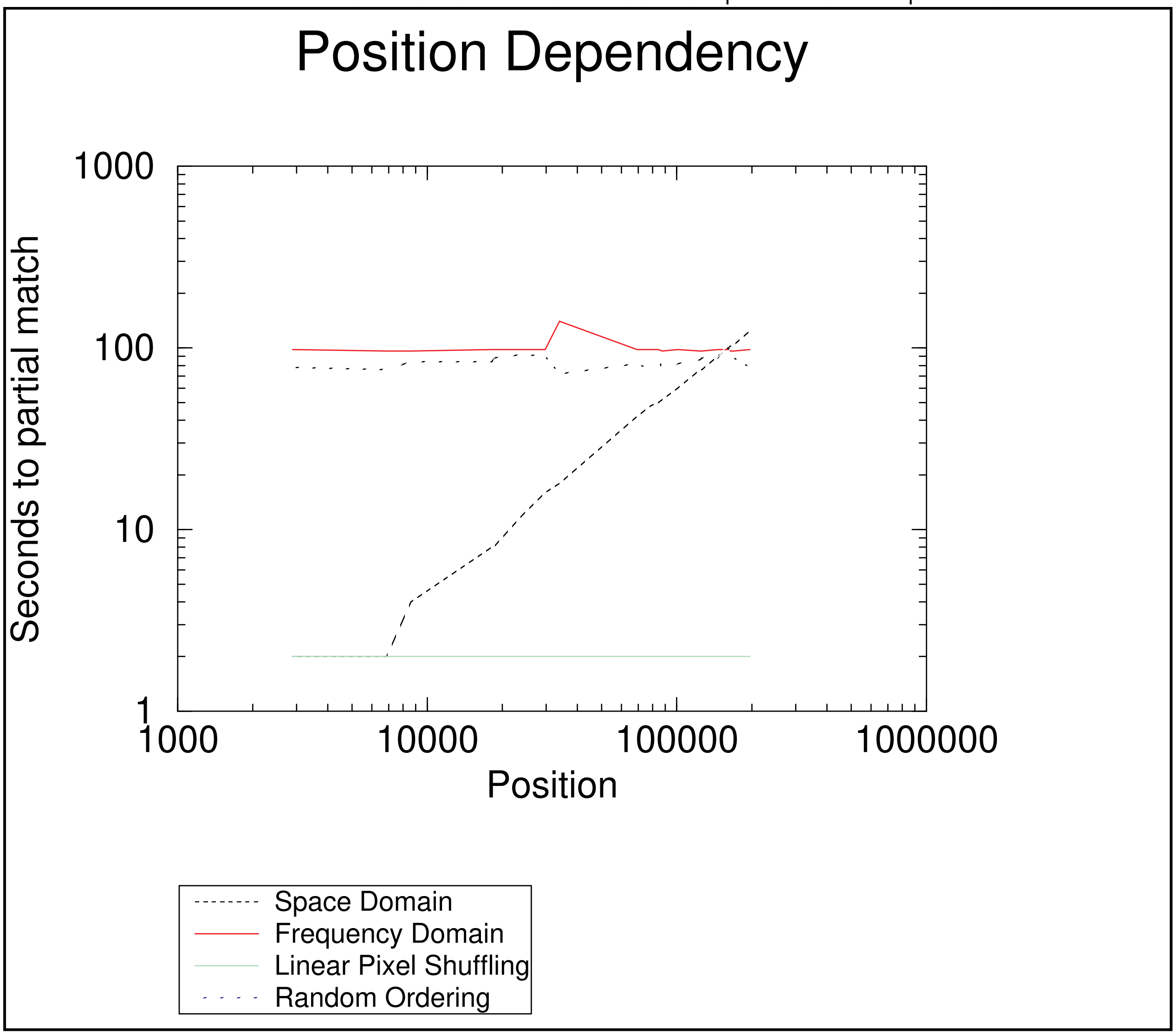


Figure 13 on page 25 shows the position vs. the number of seconds it took an algorithm to find a partial match. Position is in terms of sequential order (i.e. for a 500x500 pixel image, pixel (1,1) is the 501th pixel). This graph shows LPS, Frequency Domain, and Random Ordering to be position invariant (with LPS being the quickest of the three). Space Domain is shown to be dependent on position.

6.3 Experiment 3

TABLE 11. timing for Experiment 3 case

Case	# pixels	Seconds/ Pixel	Pixels/ Second	# of pixels til 1st partial match	Seconds
text	314,061	4.6188E-02	21.6504	2,089	9.6488E+01

The descriptions for the columns are the same as for the table in the first section. Comparing the number of seconds in column 6 (~96.5 seconds) to the number of seconds LPS took to find a partial match in Table 5 on page 20 (98 seconds), we see that the values for LPS in Table 5 are indeed correct.

6.4 Space Domain

The results for Space Domain varied depending upon the location of the pattern within the image and the pattern size. If the pattern was located near the starting position, Space Domain had a better chance of being the first to find the pattern. A bigger pattern size meant more calculations per correlation value.

6.5 Frequency Domain

Frequency Domain had good results (compared to Space Domain) when the ratio between image and pattern was small. That is, the pattern was a large fraction of the image. It was consistent for Experiments 1 and 2 when the size of the image didn't change. This was due to the fact that for a given image size, Frequency Domain will always have to do the same number of operations and so the time it takes should be consistent. Timing results were independent of position and pattern size since the timing of this algorithm is based on the number of operations performed. The number of operations is based on the number of pixels (power of 2 size) in which it has to transform.

6.6 Random Ordering

As is the case with random numbers, Random Ordering sometimes did good, sometimes did bad. It turns out to be shift and pattern size invariant. This would be expected due to its random nature.

6.7 Linear Pixel Shuffling

In all cases, LPS proved to find a partial match first. The following table shows the speed ratios for each case in experiment 1 for finding a partial match.

TABLE 12. speed ratio (in seconds) for Experiment 1 for finding a partial match

Case	Space/LPS	Freq/LPS	Random/LPS
3x3	6	50	43
5x5	14	49	42
11x11	64	50	44
41x41	792	49	50
81x81	2712	50	37

Since the pattern size is the same in all cases for Experiment 2, the timing difference can be averaged. On average, LPS found a partial match 47.7, 97.4, and 82.5 seconds faster than Space Domain, Frequency Domain, and Random Ordering, respectively for finding a partial match.

TABLE 13. speed ratio (in seconds) for Experiment 2 for finding a partial match

Case	Space/LPS	Freq/LPS	Random/LPS
(102,17)	2	48	42
(148,157)	24	49	38
(182,59)	8	49	45
(196,36)	4	49	42
(202,248)	38	48	43
(226,327)	51	48	43
(290,201)	30	49	41
(293,138)	21	49	41
(354,168)	25	49	46
(368,291)	45	49	44
(374,13)	1	48	38
(387,5)	1	49	39
(408,67)	9	70	36
(432,338)	52	48	44
(444,251)	38	48	44
(447,36)	4	49	44
(466,392)	62	49	39
(489,174)	26	48	37
(63,48)	6	49	46
(91,321)	50	49	53
Averages	24.85	49.7	42.25

Below are the difference values for Experiment 3 for finding a partial match.

TABLE 14. speed ratio (in seconds) for Experiment 3 for finding a partial match

Case	Space/LPS	Freq/LPS	Random/LP
text	104	6	9

7.0 Conclusions

LPS was able to find the pattern much quicker than the other algorithms most of the time. Within 2 seconds it was able to scan the 500x500 pixel image and match a part of the pattern (experiments 1 and 2). Results show LPS to be 10 to 5,422 seconds faster for changes in pattern size, 47.7 to 82.5 seconds faster depending on pattern position, and 452 to 10,088 seconds faster for a ***real*** test case. It is pattern size invariant and independent about position.

This project did not investigate the effects of image/pattern noise. However, the single real test case examined suggests that LPS may yield similar processing speed improvements when noise is present.

Despite the fact that LPS took longer to find the exact center (in some cases), LPS is useful as a subpattern searching algorithm. Intelligent processing can be added to allow LPS to find the center much faster than is shown here. LPS has consistently demonstrated that it has the capability to find a pattern much quicker than the other algorithms in this project.

The work done here in no way shows the full potential of LPS. In a real application, once the pattern's neighborhood is found, there would be no need to continue looking at every pixel as was done here. This experiment merely lays down the foundation to show that LPS is an algorithm that can be made use of for subpattern searching.

7.1 Limitations

LPS calculates correlation the same way as Space Domain and as such can get fooled. If any noise has a better correlation than the pattern, LPS can mistake the noise for the pattern. Any intelligence than can help Space Domain avoid false matches may also help LPS (such as noise reduction filters, thresholding correlation values, etc.). Noise reduction is beyond the scope of this project.

One problem seen with LPS is when there is an unknown number of patterns. LPS's strength is the ability to find a match quickly. When the number of patterns is not known in advance, the question arises as to when LPS should stop searching.

8.0 Future Research

As was mentioned before, LPS sometimes takes longer to find the exact center. This can be improved on. This experiment takes a simplistic approach and continues to check every pixel, even after finding a partial match.

One way to improve performance is to zero in when that first partial match is found. Instead of continuing, concentrate on the specific area the match was found by doing the LPS algorithm on this sub-sample.

Another way to speedup performance is to improve the correlation calculation. Any way to speed up the calculation would speedup the whole process.

Noise was not really an issue in this project. The first 2 experiments were noise-free. One direction to go with LPS research is to look into how it handles noise. Given the first 2 experiments with noise, the same results would be expected. It's with intelligent processing that noise becomes more troublesome. What happens when you get some correlation? Should LPS investigate the sub-sample? Or is the correlation only noise and LPS would be wasting its time? Also, if LPS is trying to zero in on the center, how long should it process before it realizes that the pattern isn't in the sub-sample? One way to get around these problems would be to set a threshold. Any correlation above this threshold would be considered pattern, anything below would be noise.

In the case of an unknown number of patterns, an intelligent processing algorithm to look into is to setup the LPS parameters based on the size of the pattern so that you are likely to hit a pattern (if one exists) on every jump. One pass through the image would tell you the correlation hot spots to zero in on.

There are several different methods to find parameters for LPS. Another area to look into is to discover which method produces the best parameters.