University of Nevada, Reno

Super-Resolution Enhancement From Multiple Overlapping Images: A Fractional Area Technique

A thesis submitted in partial fulfillment of the
requirements for the degree of Master of Science in
Geology

by

Joshua A. Michaels

Dr. James R. Carr / Thesis Advisor

August 2011

THE GRADUATE SCHOOL

We recommend that the thesis
prepared under our supervision by

JOSHUA ALAN MICHAELS

entitled

Super-Resolution Enhancement From Multiple Overlapping Images:
A Fractional Area Technique

be accepted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

James R. Carr, Ph.D., Advisor

James V. Taranik, Ph.D., Committee Member

Katharine L. DeBoer, D.M.A., Graduate School Representative

Marsha H. Read, Ph. D., Associate Dean, Graduate School

August, 2011

Abstract

With the availability of large quantities of relatively low-resolution data from several decades of space borne imaging, methods of creating an accurate, higher-resolution image from the multiple lower-resolution images (i.e. super-resolution), have been developed almost since such imagery has been around. The fractional-area super-resolution technique developed in this thesis has never before been documented. Satellite orbits, like Landsat, have a quantifiable variation, which means each image is not centered on the exact same spot more than once and the overlapping information from these multiple images may be used for super-resolution enhancement. By splitting a single initial pixel into many smaller, desired pixels, a relationship can be created between them using the ratio of the area within the initial pixel. The ideal goal for this technique is to obtain smaller pixels with exact values and no error, yielding a better potential result than those methods that yield interpolated pixel values with consequential loss of spatial resolution. A Fortran 95 program was developed to perform all calculations associated with the fractional-area super-resolution technique. The fractional areas are calculated using traditional trigonometry and coordinate geometry and Linear Algebra Package (LAPACK; Anderson et al., 1999) is used to solve for the higher-resolution pixel values. In order to demonstrate proof-of-concept, a synthetic dataset was created using the intrinsic Fortran random number generator and Adobe Illustrator CS4 (for geometry). To test the real-life application, digital pictures from a Sony DSC-S600 digital point-and-shoot camera with a tripod were taken of a large US geological map under fluorescent lighting. While the fractional-area super-resolution technique works in perfect synthetic conditions, it did not successfully produce a reasonable or consistent

solution in the digital photograph enhancement test. The prohibitive amount of processing time (up to 60 days for a relatively small enhancement area) severely limits the practical usefulness of fraction-area super-resolution. Fractional-area super-resolution is very sensitive to relative input image co-registration, which must be accurate to a sub-pixel degree. However, use of this technique, if input conditions permit, could be applied as a "pinpoint" super-resolution technique. Such an application could be possible by only applying it to only very small areas with very good input image co-registration.

In Memoriam

Dr. James Taranik (April 23, 1940 – June 21, 2011)

A great man with whom I regret not known better.

I am honored to have had him on my committee.

Acknowledgements

- Dr. James Carr, advisor
- Committee members
  - Dr. James Taranik
  - Dr. Katharine DeBoer
- For healthy discussions
  - Dr. Fred Kruse, Amie Lamb, Jay Goldfarb, Ross Whitmore, Julie Brown, Julie Johnson, Charlene Michaels
- Family and friends who kept me sane

Table of Contents

Abstract i
In Memoriam iii
Acknowledgements iv
Table of Contents v
List of Tables vi
List of Figures vii
Introduction 1
Fractional-Area Methodology 3
Proof-of-concept using synthetic, perfectly aligned data 7
Application to Digital Data 14
Conclusions 20
References 23
Appendix A 24
Appendix B 44

List of Tables

Table A: Input parameters..16

List of Figures

Figure 1: VPLR schematic diagram 1
Figure 2: Schematic pixel geometry....3
Figure 3: Synthetic IP data 8-11
Figure 4: DP data....12
Figure 5: DP value histogram....13
Figure 6: Digital input data #3 example....14
Figure 7: Staggered versus rotated pixels....15
Figure 8: DP size determination....16
Figure 9: Example section....17
Figure 10: DP maximum extent 18
Figure 11: IP/DP ratio histogram 19
Figure 12: DP extent with 2 days of processing....20
Figure 13: Camera lens-derived distortion....21
Figure A1: Diagrammed explanation of Subroutine PolyDiagPnt....30

Introduction

With the availability of large quantities of relatively low-resolution data from several decades of space borne imaging, methods of creating an accurate, higher-resolution image from the multiple lower-resolution images, called super-resolution, have been developed almost since such imagery has been around. Techniques, such as non-uniform interpolation (Ur and Gross, 1992) that uses either a direct or iterative solution, or frequency domain analysis (Tsai and Huang, 1984) that processes data in the frequency domain before returning to the spatial domain, have been proposed and tested. More recently, variable-pixel linear reconstruction (VPLR) was introduced based on using the overlapping pixel information to interpolate higher-resolution pixels. The VPLR method is similar to the fractional-area technique of this thesis (Fruchter and Hook, 2002; Merino and Núñez, 2007; Figure 1).

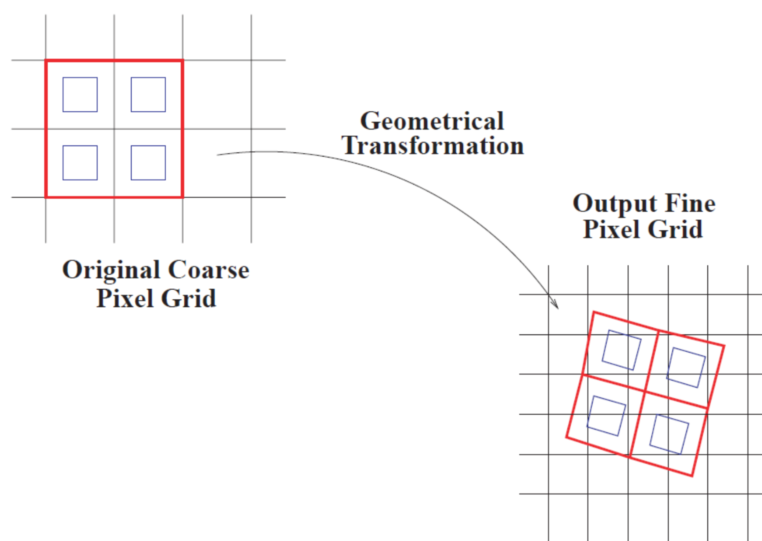


Figure 1: VPLR schematic diagram (from Fruchter and Hook, 2002).

As a prime example, Landsat has been obtaining low-resolution (30m and 60m) data approximately every 16 days since 1972 in a 10:30 polar orbit. Even after removing

cloudy images and selecting a particular season (to minimize changes due to the angle of sunlight, snow, water levels, fires, etc.), a significant amount of image data viable for super-resolution is available.

The fractional-area super-resolution technique developed in this thesis has never before been documented. Satellite orbits, like Landsat, have a quantifiable variation, which means each image is not centered on the exact same spot more than once. The overlapping information from these multiple images may be used for super-resolution enhancement by splitting a single initial pixel into many smaller, desired pixels and creating a relationship between them using the ratio of the area within the initial pixel. Because the technique stays in the spatial domain, it is potentially very computationally intense but modern 64-bit computers may be able to make it work. The ideal goal for this technique is to obtain smaller pixels with exact values and no error, a better approach than those methods that yield interpolated pixel values with consequential loss of spatial resolution.

Fractional-Area Methodology

Each initial pixel (IP) of every image will have a unique equation of fractional desired pixel (DP) areas normalized by the IP area. If the number of IPs (and therefore IP equations) is greater or equal to the number of DPs, the values of the higher-resolution desired pixels can be determined. This concept is best illustrated by the following example.

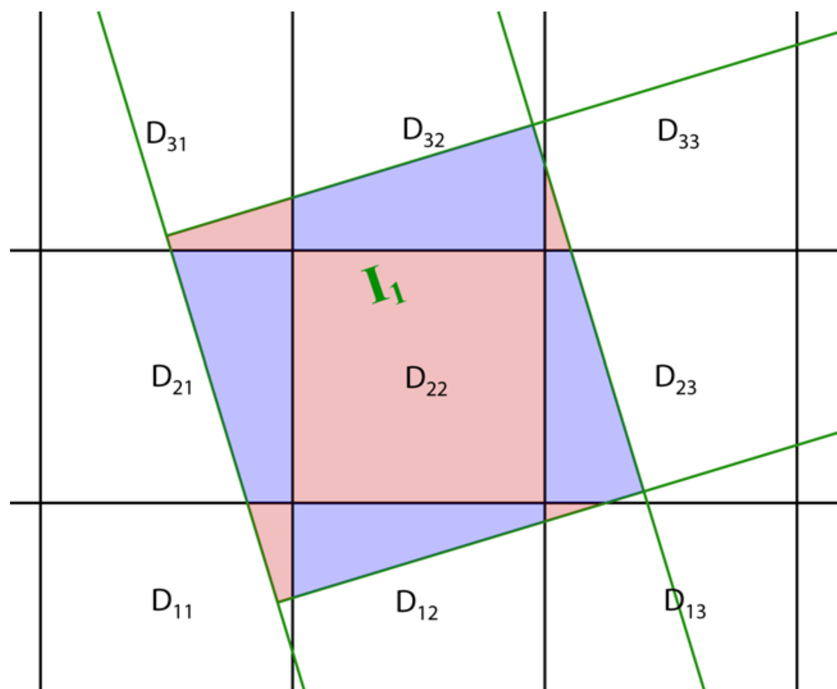


Figure 2: Two grids, the IP grid (green lines) and the DP grid (black lines), and the fractional DP areas (D_{xy}; blue/red for contrast) contained within a single IP (I_1). The DP size is 20 meters square and the IP size is 30 meters square; registration of the IP grid to the DP grid explained in text.

Figure 2 illustrates a DP grid (D_{xy}; where x and y are index integers) of 9 pixels with an IP grid overlain. All desired and initial pixels have independent values within some range, typically between 0 and 100 (grayscale). The bottom left corner vertex coordinates, size of the IP, and IP grid rotation angle relative to the DP grid are required to calculate the 9 red/blue fractional DP areas (see Appendix A for details). To construct

each unique IP equation, each fractional DP area is normalized by the IP area (30^2 = 900 m^2 in this example). Since the fractional DP area of D_{11} (red portion) is 36 m^2, the normalized fractional DP area for D_{11} is 0.04 (36 m^2/900 m^2). In other words, 4% of the value of I_1 is represented by the value of D_{11}. The full equation with all 9 fractional DP areas is shown in Equation A.

$$\frac{36}{30^2}D_{11}+\frac{90}{30^2}D_{12}+\frac{18}{30^2}D_{13}+\frac{117}{30^2}D_{21}+\frac{20^2}{30^2}D_{22}+\frac{90}{30^2}D_{23}+\frac{45}{30^2}D_{31}+\frac{99}{30^2}D_{32}+\frac{9}{30^2}D_{33}=I_1$$
$$\Downarrow \tag{A}$$
$$0.04D_{11}+0.10D_{12}+0.02D_{13}+0.13D_{21}+0.44D_{22}+0.10D_{23}+0.05D_{31}+0.11D_{32}+0.01D_{33}=I_1$$

Equation A can be expressed in the matrix form, [**A**][D_{xy}]=[I_n], as seen in Equation B, which also includes placeholders (#) for n number of unique IP equation values with the 9 common DPs.

$$\begin{bmatrix} 0.04 & 0.10 & 0.02 & 0.13 & 0.44 & 0.10 & 0.05 & 0.11 & 0.01 \\ \# & \# & \# & \# & \# & \# & \# & \# & \# \\ \# & \# & \# & \# & \# & \# & \# & \# & \# \\ \# & \# & \# & \# & \# & \# & \# & \# & \# \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ \# & \# & \# & \# & \# & \# & \# & \# & \# \end{bmatrix} \times \begin{bmatrix} D_{11} \\ D_{12} \\ D_{13} \\ D_{21} \\ D_{22} \\ D_{23} \\ D_{31} \\ D_{32} \\ D_{33} \end{bmatrix} = \begin{bmatrix} I_1 \\ I_2 \\ I_3 \\ I_4 \\ \vdots \\ I_n \end{bmatrix} \tag{B}$$

If the value of n is greater than or equal to the number of DPs, the values of the desired pixels (D_{xy}) can be directly solved from a modified form of Equation B, where k (a combination of all or some of n IP equations) equals the number of DPs. If n is comprised of 11 IP equations, {1 2 3 4 5 6 7 8 9 10 11}, k must have 9 IP equations

(equal to the number of DPs) from set *n*, say {1 3 4 5 7 8 9 10 11}, so that Equation C can be solved.

$$\begin{bmatrix} 0.04 & 0.10 & 0.02 & 0.13 & 0.44 & 0.10 & 0.05 & 0.11 & 0.01 \\ \# & \# & \# & \# & \# & \# & \# & \# & \# \\ \# & \# & \# & \# & \# & \# & \# & \# & \# \\ \# & \# & \# & \# & \# & \# & \# & \# & \# \\ \# & \# & \# & \# & \# & \# & \# & \# & \# \\ \# & \# & \# & \# & \# & \# & \# & \# & \# \\ \# & \# & \# & \# & \# & \# & \# & \# & \# \\ \# & \# & \# & \# & \# & \# & \# & \# & \# \\ \# & \# & \# & \# & \# & \# & \# & \# & \# \end{bmatrix} \times \begin{bmatrix} D_{11} \\ D_{12} \\ D_{13} \\ D_{21} \\ D_{22} \\ D_{23} \\ D_{31} \\ D_{32} \\ D_{33} \end{bmatrix} = \begin{bmatrix} I_1 \\ I_3 \\ I_4 \\ I_5 \\ I_7 \\ I_8 \\ I_9 \\ I_{10} \\ I_{11} \end{bmatrix} \quad \text{(C)}$$

However, the vectors (columns) of [**A**] must be linearly independent in order to solve for the values of the desired pixels (D_{xy}). Linear independence is established when each of the vectors of [**A**] cannot be created by any combination of any other vector multiplied by a scalar value. Practically, Equation C can be solved when the determinant of [**A**] is not zero.

Due to many combinations of IP equations not being linearly independent, a systematic testing of IP equation combinations must be performed. The number of potential unique combinations can be calculated from the values of *n* and *k* (Equations D and E).

$$\begin{aligned} &\frac{n!}{k!(n-k)!} = \text{total \# of combinations} \\ &n = \text{total \# of IP equations} \\ &k = \text{subset size (\# of DP unknowns)} \end{aligned} \quad \text{(D)}$$

$$\frac{11!}{9!(11-9)!} = 55 \quad \text{(E)}$$

Note that because factorials are used in Equation D, as n, k, and/or n-k increase linearly the total number of combinations will increase very rapidly.

Given that the fractional DP areas are normalized to each IP size, the IP size does not have to be consistent between input IP images or even within the same image.

Several major assumptions are made for the fractional-area super-resolution technique. Each input pixel is assumed to accurately represent the detail within it and the value is a perfect average of the detail within. The input image registration is perfect and has no error. Seasonal variations are assumed to be not present.

Proof-of-concept using synthetic, perfectly aligned data

A Fortran 95 program (see Appendix A for detailed description) was developed to perform all calculations associated with the fractional-area super-resolution technique. Double-precision numbers are used for all floating-point values (accurate up to 16 decimal places); this high precision is not explicitly needed, but helps minimize computational error. Computational error is defined as rounding errors at the end of a decimal that can be magnified to a significant degree with a sufficient number of calculations acted upon it. The fractional areas are calculated using traditional trigonometry and coordinate geometry. Linear Algebra Package (LAPACK) is a freely-available software package for solving many linear algebra problems with efficient algorithms (Anderson *et al.*, 1999), and is used to solve for the DP values.

In order to demonstrate proof-of-concept, a synthetic dataset was created using the intrinsic Fortran random number generator and Adobe Illustrator CS4. Seven 3x3 input images were created using Illustrator and the bottom-left corner coordinates and the angle of rotation were recorded for use as input parameters of the Fortran program (Figure 3). A 7x7 grid of randomly-generated DP values (between 0 to 100) was created to derive the IP values (from the fractional area relationships) and to have a “correct” value with which to compare the program’s solutions (Figure 4). These derived IP values and the associated fractional areas were then solved for the DP values.

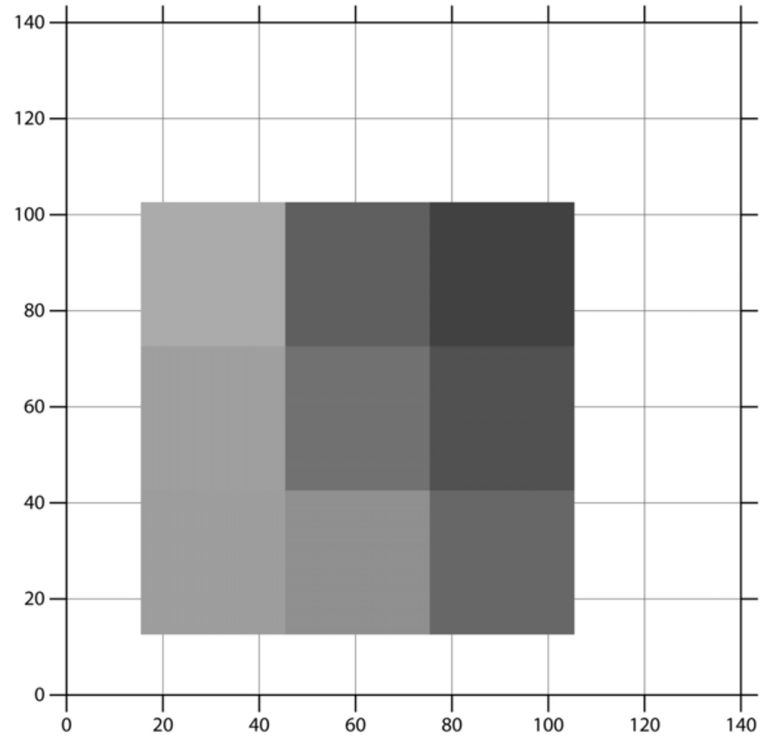


Figure 3: Seven 3x3 input images. The IP values were calculated from DP values using the fractional area relationships.

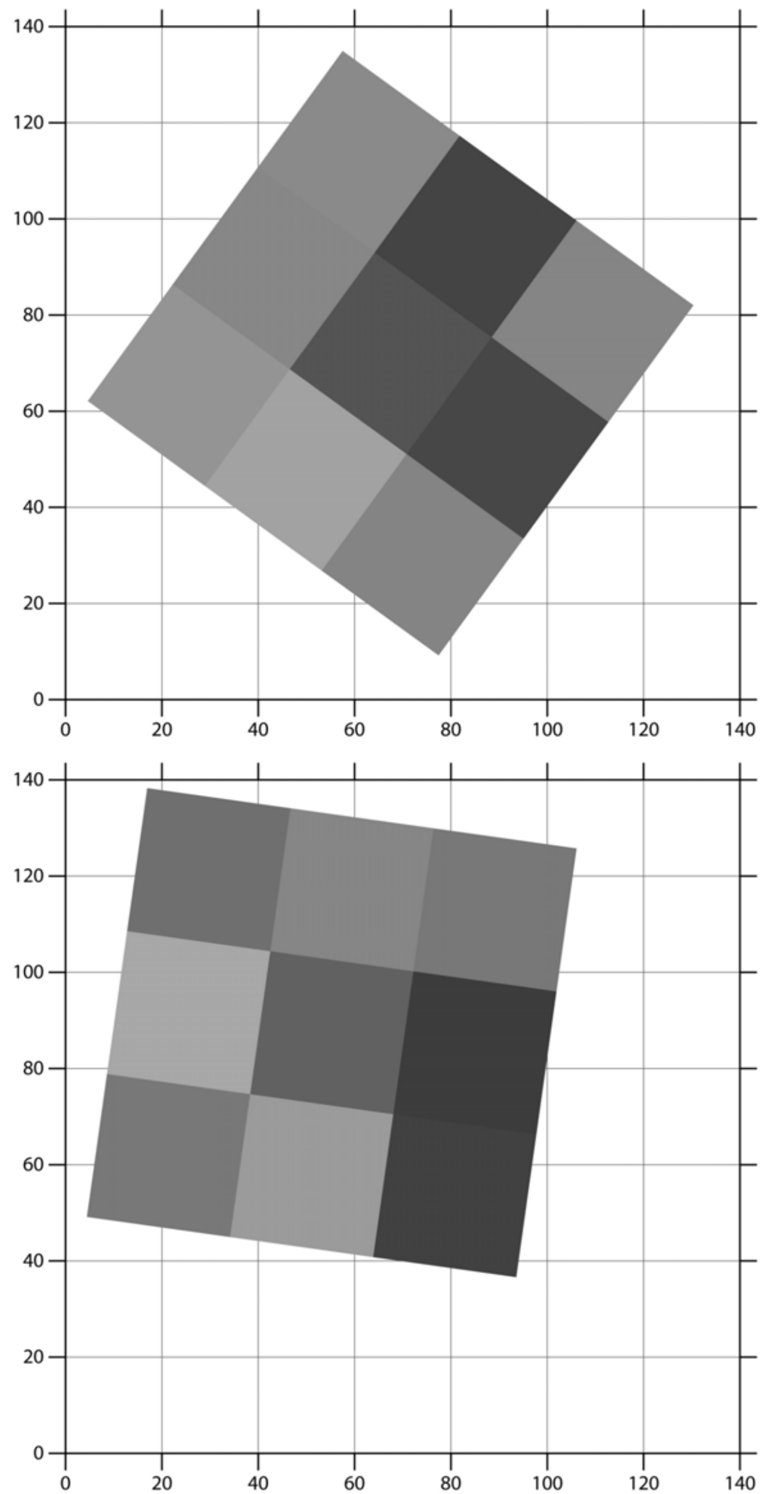


Figure 3 (continued): Seven 3x3 input images. The IP values were calculated from DP values using the fractional area relationships.

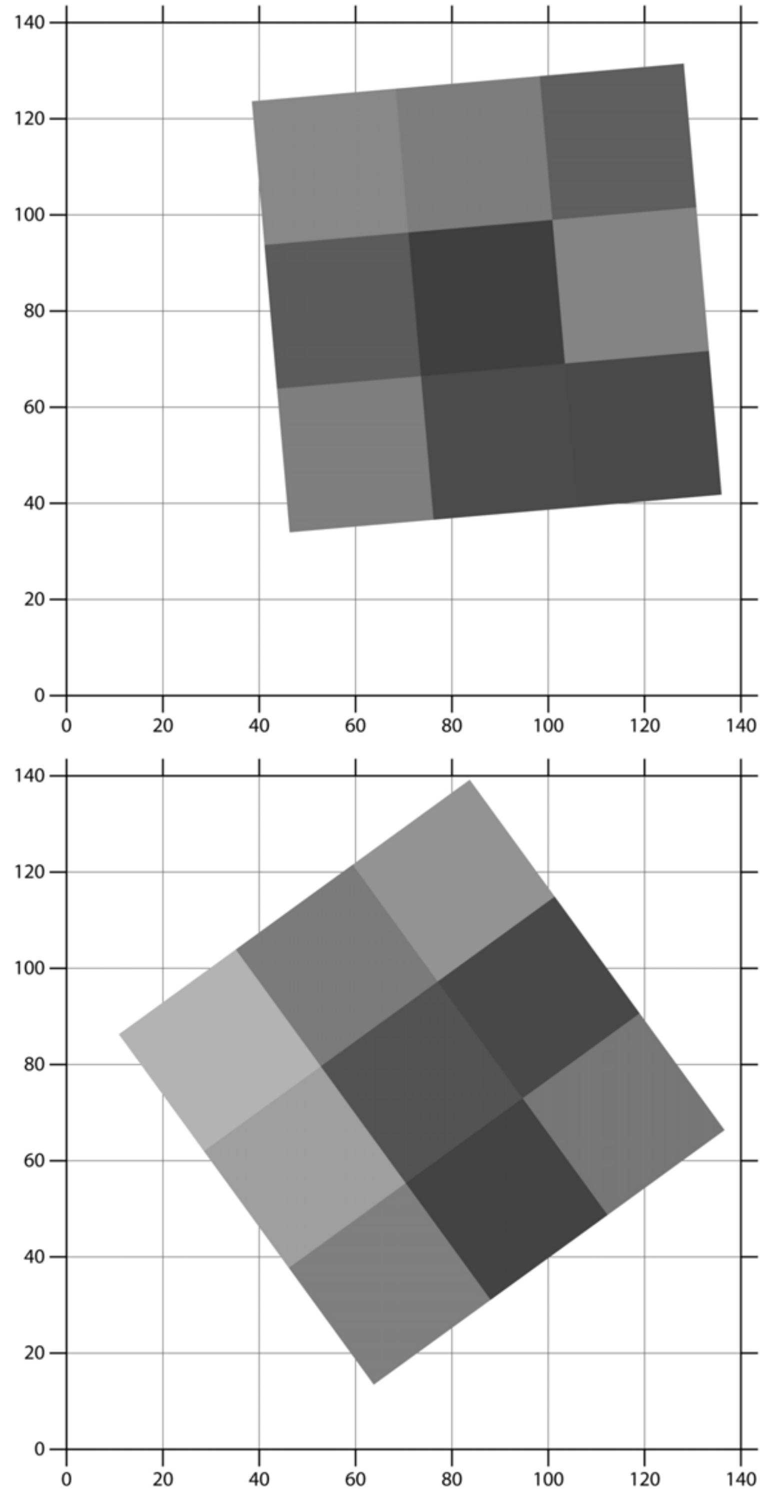


Figure 3 (continued): Seven 3x3 input images. The IP values were calculated from DP values using the fractional area relationships.

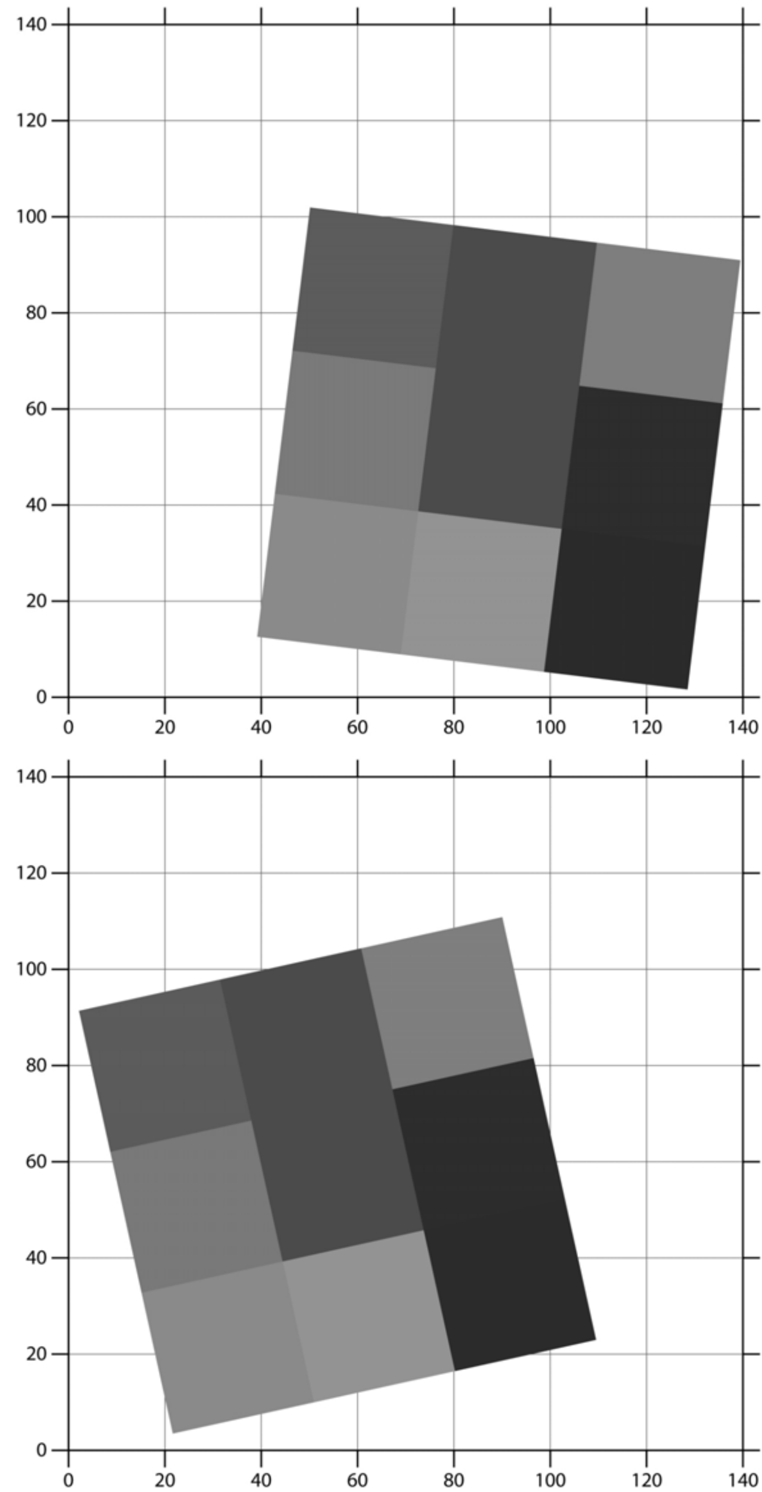


Figure 3 (continued): Seven 3x3 input images. The IP values were calculated from DP values using the fractional area relationships.

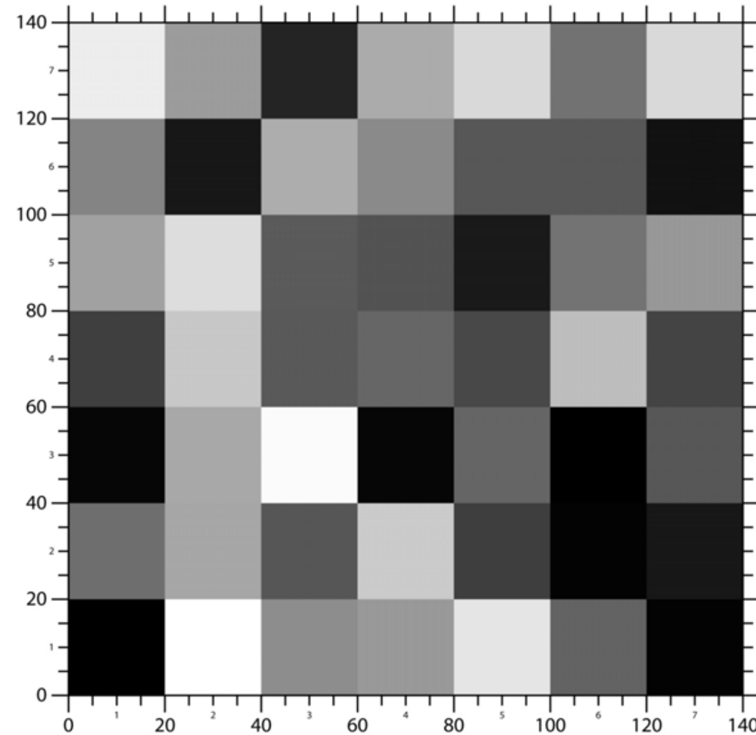


Figure 4: The initial DP image whose values are randomly-generated. Also is the final output of the Fortran program.

Due to the linear independence problem, 10 million combinations (out of a possible ~37.4 trillion) were tried resulting in 456,365 reasonable solutions. Some computational error and erroneous solutions arose, but greater than 94.9% of solutions were confined to within 1×10^{-8} of the correct DP value (see example Figure 5).

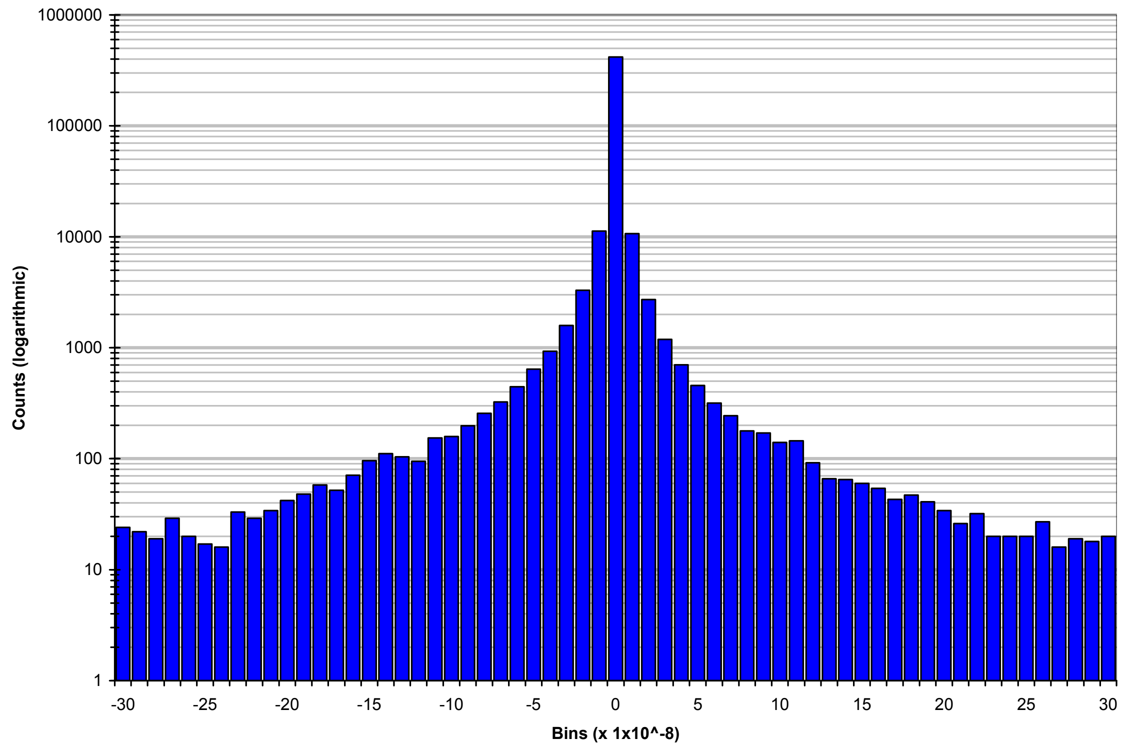


Figure 5: Histogram for an example DP illustrating the difference between the synthetic dataset solutions and the true values; this is the most spread-out answer (bin size = $1x10^{-8}$m centered about zero; 99.8% of total counts shown).

Application to Digital Data

In order to test the real-life application of the fractional-area super-resolution technique, a Sony DSC-S600 digital point-and-shoot camera with a tripod was used to image a large US geological map under fluorescent lighting. Pure parallel rotation was obtained using a feature of the tripod physically rotating the camera without moving the tripod base. Moreover, the camera was positioned parallel to the geological map so that rotation would be pure parallel rotation. Multiple images were taken from each rotation angle. A single image from each rotation angle was chosen by considering and minimizing lighting differences and blur. A set of 5 images was chosen and used for the fractional-area super-resolution technique.

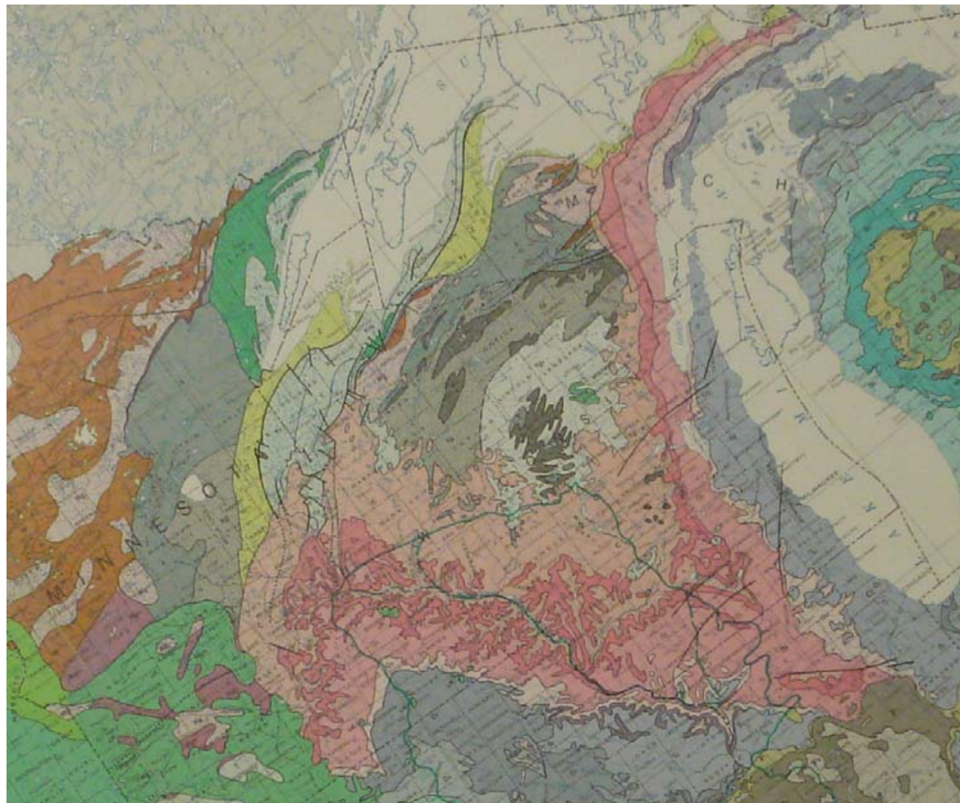


Figure 6: Input image 3 as an example.

The images were imported into Adobe Illustrator CS4 and two control points were chosen that were common to all 5 images. Adobe Illustrator CS4 was chosen to minimize image post-processing and the ease of overlaying another grid and obtaining coordinates from it. These control points were used for relative referencing of the 5 images and to determine the amount of rotation from a common baseline, which was determined by overlaying a desired pixel (DP) grid over the images. The images were cropped to 1200x1000 pixels (Figure 6). The bottom-left corner and rotation of the cropped images relative to the DP grid was recorded for use in the program. The cropped 1200x1000 images were rotated to the baseline (horizontal) and saved as 24-bit bitmap images. This last geometric correction was needed because the pixels of the rotated images are not rotated, just staggered (Figure 7).

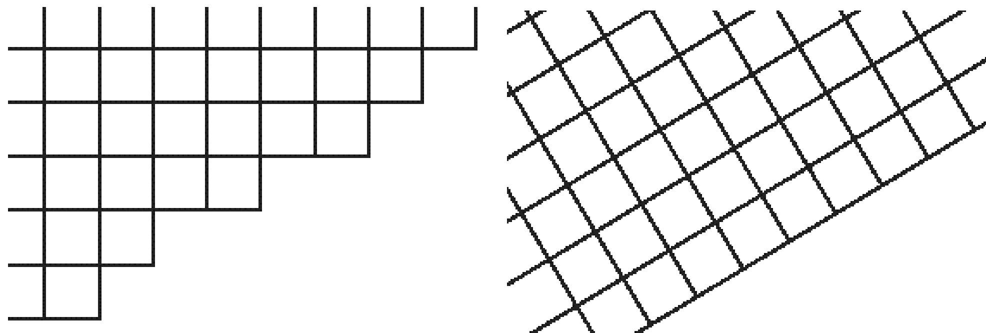


Figure 7: Difference between staggered (left) and rotated (right) pixels

Because it would be very difficult to calculate the image pixel size relative to the map itself, the image pixel size was set to 1. The lack of unit does not affect the program because the ratio between the input image pixel size and the desired pixel size is the crucial component. However, minor differences in the distance from the camera to map as the pictures were taken (and therefore from image to image) could affect the program

result since the pixel sizes are assumed to be identical. See Table A for input parameters used.

Bottom Left Corner X	209.115	143.000	17.747	19.502	62.947
Bottom Left Corner Y	285.029	560.000	696.094	927.620	1088.735
Rotation Angle	2.033	-23.407	-31.716	-48.764	-61.029
IP Grid Size	1.0	1.0	1.0	1.0	1.0
Number of IP X	1200	1200	1200	1200	1200
Number of IP Y	1000	1000	1000	1000	1000
DP Grid Size	0.6				

Table A: Input parameters

The desired pixel size was determined from a simple program that calculates the number of desired pixels at a certain resolution intersecting with at least one initial pixel (IP) from the input images. The IP/DP size ratio was incrementally decreased and the difference between the number of IPs and number of DPs was recorded. The resulting table (Figure 8) shows that a DP size of 0.6 would allow a reasonable extra number of IPs to compensate for the linear dependency problem.

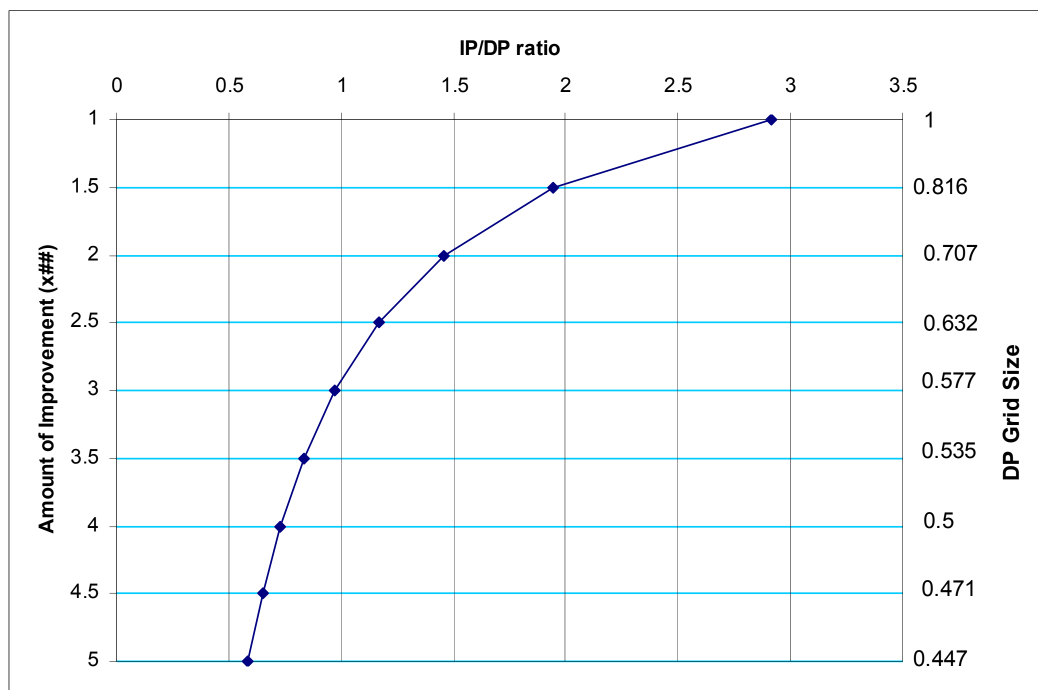


Figure 8: DP size determination based on the input parameters. The IP/DP ratio chosen was just above 1 to allow for overdetermined matrices.

In order to realistically solve a 2708x2604 grid of DPs, everything must be divided into manageable pieces. Consideration into choosing a section size must consider the DP matrix size to be solved. If the section size is too large the linear algebra solver would bog down when processing thousands of combinations because the maximum DP matrix size for a given section size is determined by squaring the section size. Therefore section sizes of 5, 10, 15, and 20 desired pixels would correspond with DP matrices of 25x25, 100x100, 225x225, 400x400. Because this was a somewhat subjective choice, a section size of 10 was chosen as the best balance between section size and performance (Figure 9).

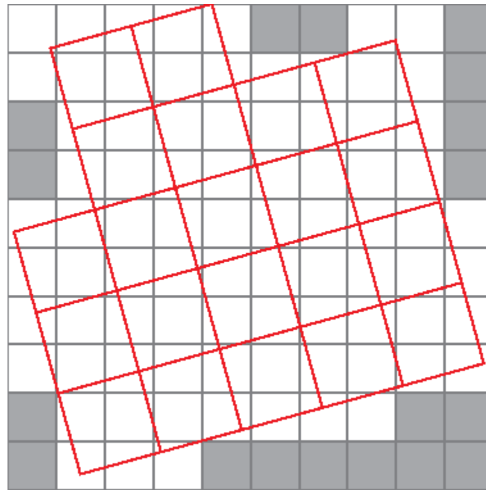


Figure 9: An example section. The "x"-ed out boxes are DP pixels that cannot be solved for since they are not covered by an IP.

The input images were preprocessed to calculate two computationally intensive items: a list stating the presence of a DP covered by an IP and another list of IP vertices (Figure 10).

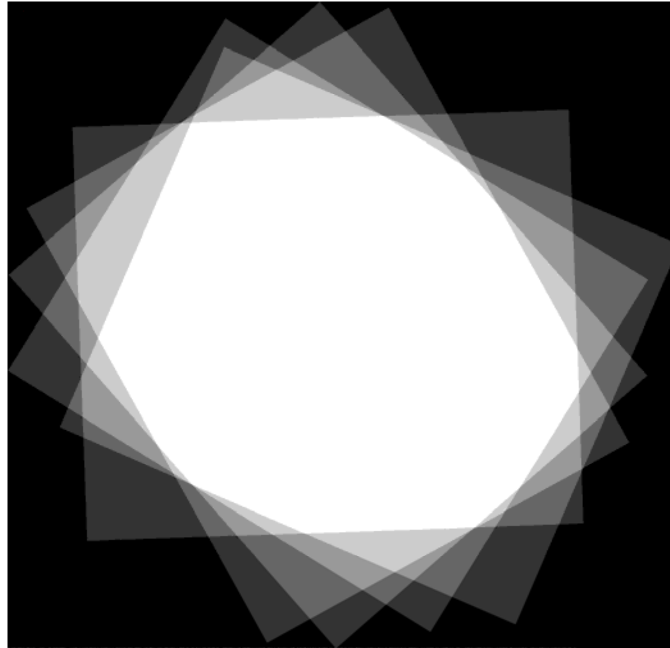


Figure 10: All DP pixels that are covered by at least one IP. (black = no IP coverage ➔ white = complete IP coverage by all 5 input images)

To minimize unnecessary calculations and wasted time, the program was cycled to the next section if the preprocessor-derived list of used DPs has no DPs for the given section, a given section's IP/DP ratio is less than 1, or no solutions are obtained for the first color (blue) of the RGB imagery. Each of the three color components were solved separately while using the same fractional area relationships; however, only if blue had a successful solution would the other two colors be solved. The number of IP equation combinations was subjectively set to 100,000 combinations as the best balance between thoroughness and performance.

The five 1200x1000 input images created a total DP grid of 2708x2604 pixels. The DP sections were overlapped by the diagonal length of an initial pixel to make certain that all DPs were covered. This created 143,592 (386x372) separate sections, of which only 49,990 (34.8%) had an IP/DP ratio greater or equal to 1 (Figure 11). Each calculation of the fractional areas and a set of combinations took ~120 seconds for all

three colors and ~60 seconds if blue did not have any solutions (using one processor from an AMD Phenom X4 9500 2.2GHz 64-bit processor with 4GB RAM running Ubuntu Linux). This equates to processing time of ~69.4 days and ~34.7 days, respectively; very undesirable processing times for such a relatively small image.

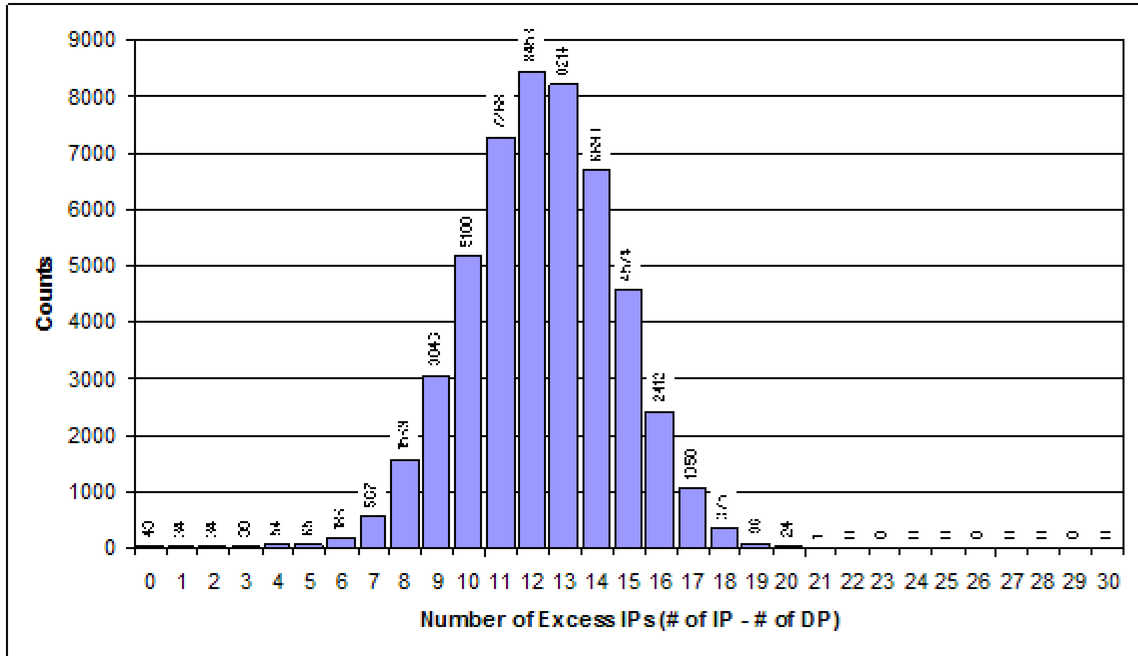


Figure 11: IP/DP ratio histogram in terms of excess number of IPs.

Solutions for each section that had an IP/DP ratio of greater than 1 were essentially any value, with no consistency. A reasonable solution would have been in the range of 0 to 255, because this is the maximum range of a single color recorded in a byte (8 bits = 2^8 different combinations = 256). By comparison, a single DP from one section would, because 100,000 IP combinations were tried, have a range of values from -125,085 to 658,054.

Conclusions

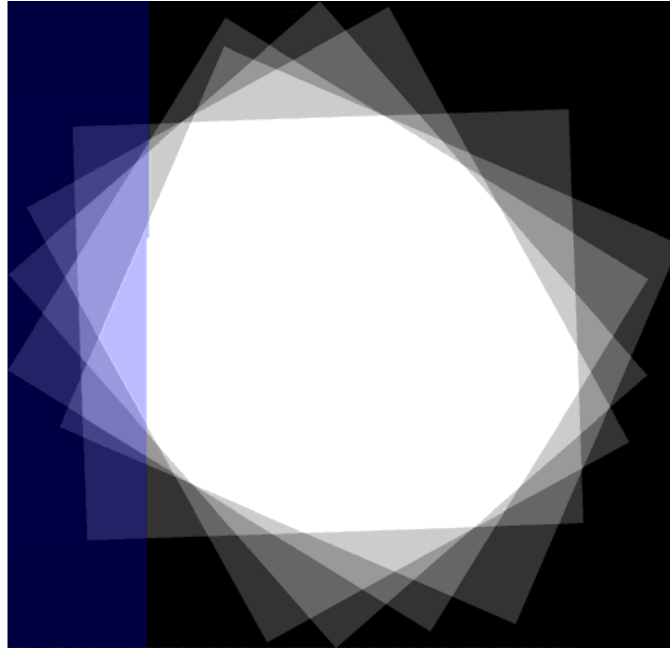


Figure 12: Blue shading indicates the progress from two days of processing.

The fractional-area super-resolution technique works in perfect synthetic conditions, but did not successfully produce a reasonable or consistent solution in the digital photograph enhancement test. The prohibitive amount of processing time (up to 60 days for a relatively small enhancement area) severely limits the practical usefulness of fraction-area super-resolution (Figure 12). Fractional-area super-resolution is very sensitive to relative input image co-registration, which must be accurate to a sub-pixel degree. However, use of this technique, if input conditions permit, could be applied as a "pinpoint" super-resolution technique. Such an application could be possible by only applying it to only very small areas with very good input image co-registration.

Several factors led to the unsuccessful enhancement of digital photography. Relative image co-registration encompasses a large number of variables that need to be accounted for. The Sony DSC-S600 used to take the input photographs has a significant

variable arcing distortion as the lens moves from wide-angle to telephoto (Figure 13). Using a camera with less distortion would increase the accuracy and decrease the level of un-distortion processing of relative image co-registration.

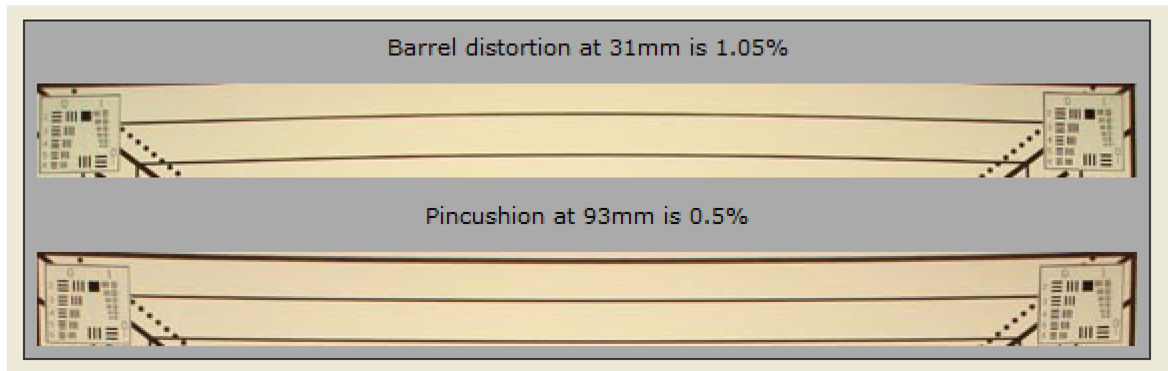


Figure 13: Camera lens-derived distortion for the Sony DSC-S600 (from http://www.imaging-resource.com/PRODS/S600/S600A.HTM)

The referencing technique used in the digital photographic experiment was proven to be much too crude, and a better referencing technique or program, such as ArcGIS or ENVI, could possibly provide a sub-pixel level of accuracy. Additionally, the angle of the lens to an object (even a flat object parallel to the camera body) would vary as the camera is rotated in any direction. There will be a "stretching" effect that would be different for every rotated image.

The value of the input images' pixels would also be a significant factor to consider in the fractional-area super-resolution technique. The details of how the imagery is processed and what approximations to the pixel values as digital rotation is applied, if any, are made in the commercially available programs like Adobe Illustrator, ArcGIS, and ENVI would be very important. Additionally, because the fluorescent lighting (standard commercial interior lighting) works by "flickering" off and on at a rate faster than the eye can distinguish, the illumination of the geological map would not be

consistent from image to image, although the relative value differences from pixel to pixel should be reasonably consistent.

References

Anderson, E., Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, D. Sorensen, **1999**, *LAPACK Users' Guide*: Society for Industrial and Applied Mathematics, 3rd edition, Philadelphia, PA, 350 p..

Fruchter, A.S. and R.N. Hook, **2002**, *Drizzle : A Method for the Linear Reconstruction of Undersampled Images*, Publications of the Astronomical Society of the Pacific, v. 114, p. 144-152.

Merino, M.T. and J. Núñez, **2007**, *Super-resolution of remotely sensed images with variable-pixel linear reconstruction*, IEEE Transactions on Geoscience and Remote Sensing, v. 45, no. 5, p. 1446-1457.

Tsai, R.Y. and T.S. Huang, **1984**, *Multipleframe image restoration and registration*: in Advances in Computer Vision and Image Processing. Greenwich, CT: JAI Press Inc., 1984, pp. 317-339.

Ur, H. and D. Gross, **1992**, *Improved resolution from sub-pixel shifted pictures*, CVGIP: Graphical Models and Image Processing, v. 54, p. 181-186.

Appendix A: Line-by-line Description of Fortran 95 Code

1) Kindtester.f95

1.1) Program KindTester

This small program is for determining the system specific numbers for real number types and integer number types ("kind numbers"). In the case of the integer number types it will demonstrate the rough range of each type (10^n format). Once these numbers are known for a specific computer they can be used correctly in any program. For maximum portability however, the following code placed in the declaration statement in a module or at the beginning of each program or subroutine should be used:

```
integer,parameter:: S = kind(0.0)   !Single precision for real
integer,parameter:: D = kind(0.0D0) !Double precision for real
integer,parameter:: B = selected_int_kind(1)      ! 1 byte
integer,parameter:: By = selected_int_kind(3)     ! 2 byte
integer,parameter:: Byte = selected_int_kind(7)   ! 4 byte
```

The above code will assign the correct kind values to unchangeable variables (S, D, B, By, Byte) for use in the rest of the program. The kind() and selection_int_kind() intrinsic functions will determine the specific computer kind values for maximum code portability.

2) RandomNumForDP.f95

2.1) Program RandomNumForDP

This program creates a file, `RandomDPgrid.txt`, and populates it with 100,000 random numbers from the intrinsic number generator subroutine, `random_number()`. `RandomDPgrid.txt` will be used later in the synthetic data experiment as the DP grid values.

3) AmtImpCalc.f95

3.1) Program IP_DP_diff

The purpose of this program is to aid in the determination of what the best resolution improvement is possible for a given set of input images.

3.1.1) <u>Lines 5 – 16</u>
Variable declarations
3.1.2) <u>Line 19</u>
Set many array sizes using `n_IPs` and fixed integers; these arrays sizes will not change within the program.
3.1.3) <u>Lines 23 – 29</u>
The initial parameters of the input images to be used throughout the program.
3.1.4) <u>Line 31</u>
Open a file, `results.txt`, to record the results. Look for `write(1,`*[formatting]*`)` statements for details of what is recorded.
3.1.5) <u>Line 34</u>
The improvement iteration where 0 is equivalent of starting at 1x (no change) and iterating the equivalent of 0.5x increments (1.5x, 2x, 2.5x, etc) to the maximum value.

3.1.6) Lines 36 – 39
Iterating through all IP input images and calculate the DP size using modifiers from Line 34 [`i_D`]. Call the `DP_int_num_calc` subroutine.
3.1.7) Line 43
Allocate memory for the rank-3 array `DP`.
3.1.8) Lines 45 – 51
Iterating through all IP input images and calculate the DP size using modifiers from Line 34 [`i_D`]. Call the `DP_refine` subroutine. Display data to the screen and record data into `results.txt`.
3.1.9) Lines 53 – 60
Iterating through the array `DP` and count the number of DPs within any IP image.
3.1.10) Lines 62 – 65
Iterating through all IP input images and count the total number of IPs.
3.1.11) Lines 67 – 68
Record and display the total number of DPs, total number of IPs, and the difference between them for the given DP size.
3.1.12) Line 70
Deallocate the array `DP` for the next DP size.

3.2) Subroutine DP_int_num_calc
A subroutine for the calculating the smallest horizontal (zero rotation) DP rectangle that can encompass a certain rotated input image.

3.2.1) Lines 88 – 91
Variable declarations: any variable labeled `IN` cannot be modified in the subroutine and any variable labeled `OUT` is the subroutine's main output.
3.2.2) Lines 94 – 95
Common mathematical phrases for later equations; idea here is to only calculate the values once and not 6 times (for performance improvement).
3.2.3) Lines 96 – 102
The array `IPcorners(n,{x,y})` assumes a rectangular shaped input image and calculates the 3 other corner points' coordinates (since the bottom-left corner is already known). See Appendix B for mathematical details.
3.2.4) Lines 105 – 107
The output array for this subroutine, `n_DP(n,{x,y})`, records the smallest horizontal rectangle of DPs that will enclose the rotated input image (n = 1,2). Additionally the actual number of DPs is calculated (n = 3).

3.3) Subroutine DP_refine
Refines the number of DPs in a enclosing rectangle from *DP_int_num_calc*; any DP completely outside the rotated input image is declared null.

3.3.1) Lines 121 – 126
Variable declarations: any variable labeled `IN` cannot be modified in the subroutine and any variable labeled `OUT` is the subroutine's main output.

3.3.2) Lines 129 – 130
Common mathematical phrases for later equations; idea here is to only calculate the values once and not 6 times (for performance improvement).
3.3.3) Lines 131 – 137
The array `IPcorners(n,{x,y})` assumes a rectangular shaped input image and calculates the 3 other corner points' coordinates (since the bottom-left corner is already known). See Appendix B for mathematical details.
3.3.4) Lines 139 – 146
Cycle through all DP vertices and determine if that point is within the input image. Array `temp(x,y)` is used to flag each vertex yea or nay.
3.3.5) Lines 148 – 155
Cycle through all DPs and, if any corner of a DP pixel was within the input image, then flag array `DP(x,y)` positively.

4) InputDataModule.f95
4.1) Module Input_data
This module contains common input data needed by most other programs and subroutines. This is the file most users will modify and tweak.

4.1.1) Line 4
The declaration `SAVE` will not allow any program or subroutine to modify the values of following variables.
4.1.2) Lines 6 – 10
Parameter declaration integers that allow for maximum portability for the program by determining the computer-dependent values for single and double precision real numbers and integer range numbers (1, 2, and 4 bytes per integer).
4.1.3) Line 12
The number of input images; this is used for following array size declarations, hence why it is declared a parameter.
4.1.4) Lines 14 – 23
These are declaration variables to be initialized by the `Subroutine Data_Initialize`. The variables are individually described with comments in the program.

4.2) Subroutine Data_Initialize
As the name implies, this subroutine initializes the module declaration variables with properties of the input image and program performance tweaking variables. No need for line-by-line description.

5) MS_Subroutines.f95
5.1) Subroutine DP_int_num
A subroutine for the calculating the smallest horizontal (zero rotation) DP rectangle that can encompass a certain rotated input image. It is very similar to *Subroutine DP_int_num_calc*. Early subroutine; possibly needs some tweaking to work with current program.

5.1.1) Lines 17 – 19
Variable declarations: any variable labeled `IN` cannot be modified in the subroutine and any variable labeled `OUT` is the subroutine's main output.
5.1.2) Lines 24 – 25
Common mathematical phrases for later equations; idea here is to only calculate the values once and not 6 times (for performance improvement).
5.1.3) Lines 26 – 32
The array `IPcorners(n,{x,y})` assumes a rectangular shaped input image and calculates the 3 other corner points' coordinates (since the bottom-left corner is already known). See Appendix B for mathematical details.
5.1.) Lines 35 – 37
The output array for this subroutine, `n_DPpixels(n,{x,y})`, records the smallest horizontal rectangle of DPs that will enclose the rotated input image (n = 1,2). Additionally the actual number of DPs is calculated (n = 3).

5.2) Subroutine Num_Vert
This subroutine calculates the number of vertices for the DP grid, IP grid, and DP/IP intersections. Early subroutine; possibly needs some tweaking to work with current program.

5.1.1) Lines 66 – 68
Calculates the number of vertices for each grid.
5.1.2) Line 70
The total number of vertices for all grids.

5.3) Subroutine DP_XY_grid
Calculates the DP grid vertices. Early subroutine; possibly needs some tweaking to work with current program.

5.3.1) Lines 98 – 105
Cycles through the X and Y direction of the DP grid's vertices.

5.4) Subroutine Intersection_IP
Calculates the IP grid intersections. Early subroutine; possibly needs some tweaking to work with current program.

5.4.1) Lines 131 – 139
Cycles through all IP grid lines and calculates vertices values.

5.5) Subroutine IP_vs_DP_intersection
Calculates the vertices where the IP grid and DP grid interect. Early subroutine; possibly needs some tweaking to work with current program and some of the X and Y variables may be in the wrong order.

5.5.1) <u>Line 166</u>
A commonly used fraction; enhance performance by only calculating it once.
5.5.2) <u>Lines 169 – 208</u>
This section is quite large, but must be considered as a single entity. The main loops using the variables `j` (Line 170) and `k` (Line 190) represent the e_j and f_i variables of Appendix B, respectively. Variable `i` is iterated by one after any calculation to populate the array `IvD_XY` with all four main equation sections, which are the following:

- Lines 172 – 176
    - Appendix B → Equation 3
- Lines 178 – 187
    - Appendix B → Equation 4
    - IF statement needed to eliminate dividing by 0
        - when $tan\ \theta = 0$
- Lines 192 – 201
    - Appendix B → Equation 5
    - IF statement needed to eliminate dividing by 0
        - when $tan\ \theta = 0$
- Lines 203 – 207
    - Appendix B → Equation 6

5.6) Subroutine All_Vertices
As the name suggests, this subroutine uses the previous subroutines to calculate all the vertices. It then combines the separate arrays into single large array (All_XY) and determines which vertices are outside the IP area. Early subroutine; possibly needs some tweaking to work with current program.

5.6.1) <u>Lines 241 – 250</u>
These lines are mainly calling other subroutines:

- *Data_Initialize* (Section 4.2)
- *Num_Vert* (Section 5.2)
    - Allocation immediately after this subroutine (Line 245) defines the array sizes for the next three subroutines
- *DP_XY_grid* (Section 5.3)
- *Intersection_IP* (Section 5.4)
- *IP_vs_DP_intersection* (Section 5.5)

5.6.2) <u>Lines 253 – 256</u>
Open a log file and write some formatted labels and variables for viewing.
5.6.3) <u>Line 259</u>
This DO loop (running until Line 314) iterates through every vertex calculated (the variable `n_vert`).
5.6.4) <u>Lines 261 – 278</u>
This IF statement combines all the individual arrays (`IvD_XY`, `Inter_XY`, `DP_XY`) into a single array (`All_XY`).

5.6.5) Lines 282 – 285
Using the Equations 3 and 5 of Appendix B, comparison numbers for the top, bottom, left, and right sides of the IP area are calculated. They will be used in the next section.
5.6.6) Lines 288 – 307
This IF statement uses the comparison values from Section 5.6.5 to determine if the vertex is within the IP area. In essence, this is simply comparing on which side of each of the four comparison lines each vertex is. If the vertex fails any one of the four tests (Line 288, 292, 296, or 300), then the vertex is not within the IP area and assigned a flag value of 0. If the vertex passes all four tests, it is assigned a flag value of 10.
5.6.7) Lines 311 – 313 and 316 – 318
More record keeping for use in other programs using the file opened in Section 5.6.2.

6) AreaFuncSubs.f95

6.1) Function PolyArea

This function calculates the area of any convex polygon given its vertices and returns a double precision value.

6.1.1) Lines 20 – 21
Allocate three arrays and initialize them.
6.1.2) Lines 24 – 30
Search for any duplicate vertices in the polygon and flagging any that are found. This step is needed since mistakes happen and duplicates will really give the wrong answer.
6.1.3) Lines 33 – 43
If there are any duplicates, they are removed from the X and Y arrays and the total number of vertices is reduced by the number of duplicates.
6.1.4) Lines 45 – 48
Testing to make sure that a polygon still exists after duplicate removal (or lack thereof)
6.1.5) Line 50
Calls the subroutine *OrderVertices* (Section 6.3); for the math to work the vertices must be in clockwise or counter-clockwise order.
6.1.6) Lines 52 – 61
The area is calculated by the equation below:

$$A = \left| \frac{x_n y_1 - y_n x_1 + \sum_{i=1}^{n-1} x_i y_{i+1} - y_i x_{i+1}}{2} \right|$$

where the vertices (x_i, y_i) are sorted in a clockwise or counterclockwise direction and n is the total number of polygon vertices. The final value is assigned to the function PolyArea (and returned to the calling program).

6.2) Subroutine PolyDiagPnt

This subroutine determines which point(s) is/are diagonally across or along a polygon edge from vertex #1. Vertex #1 is arbitrarily the first vertex in the input array; this subroutine will work with any vertex as #1.

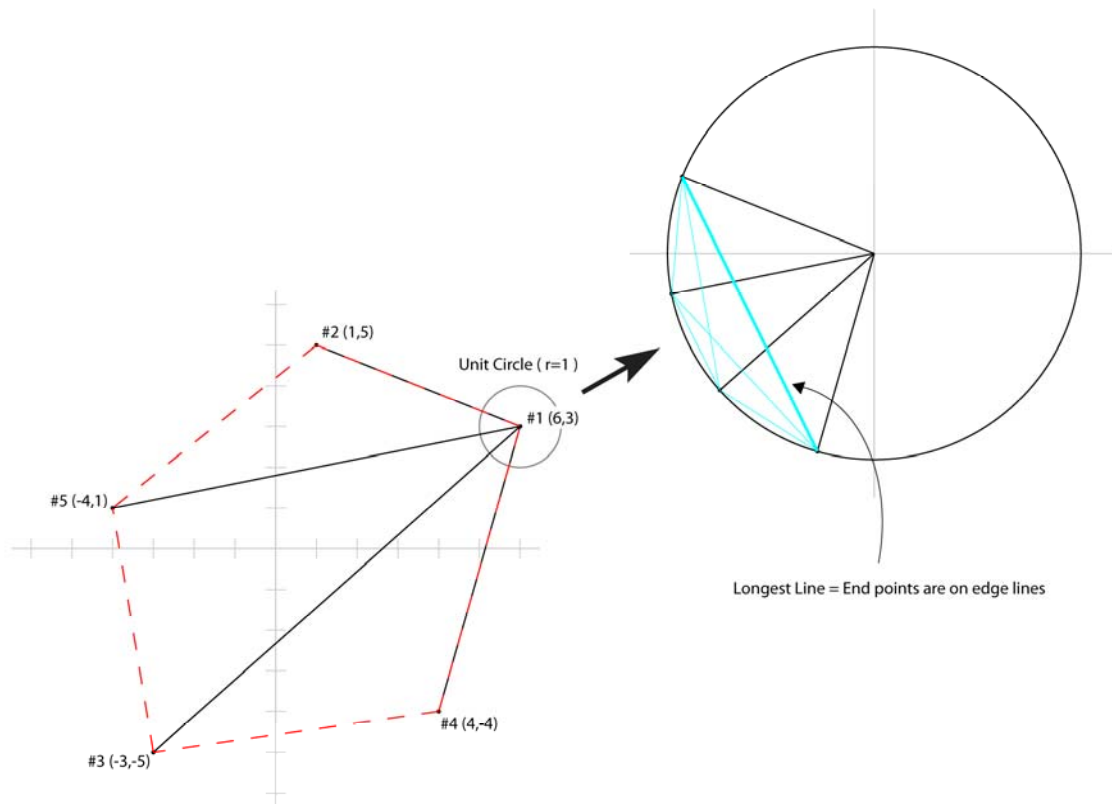


Figure A1: Diagrammed explanation of *Subroutine PolyDiagPnt*

6.2.1) Lines 90 – 98

This DO loop calculates the Cartesian coordinates of the point where each line intersects the unit circle (a circle where the radius = 1), which is centered about vertex #1. The mathematical basis is rooted in trigonometry:

$$\cos\theta = \frac{\Delta x}{l_{AB}} = \frac{x_{AB}}{1} \Rightarrow x_{AB} = \frac{\Delta x}{l_{AB}}$$

$$\sin\theta = \frac{\Delta y}{l_{AB}} = \frac{y_{AB}}{1} \Rightarrow y_{AB} = \frac{\Delta y}{l_{AB}}$$

where Δx and Δy are the difference between the x and y coordinate values of the two line endpoints, l_{AB} is the length of the line segment, and x_{AB} and y_{AB} are the coordinate values of the point where the line segment intersects the unit circle.

6.2.2) Lines 102 – 106

This DO loop calculates the number of blue lines (Figure A1) needed. The name of the main variable, `n_Angles`, refers to the use of trigonometry and could be renamed to something more explanatory.

6.2.3) Lines 108 – 120

This DO loop calculates the length of the blue lines in Figure A1.

6.2.4) Lines 122 – 124
This DO loop selects the array index of the longest blue line (thicker blue line in Figure A1).
6.2.5) Lines 126 – 136
This DO loop fills the output arrays. In the case of a triangle, it is possible that the array `Diag()` will not be populated. However, while it does work properly for triangles, some code modifications could be made to make it more portable with other compilers (such as making `Diag()` an optional output value).

6.3) Subroutine OrderVertices
As the name suggests, this subroutine will put vertices in a clockwise or counter-clockwise order around a convex polygon.

6.3.1) Lines 159 – 160
Place the input arrays `X()` and `Y()` into temporary arrays for processing.
6.3.2) Lines 166 – 169
Populate the array `Corr_ind()` with initial values used for indicating the original vertex placement.
6.3.3) Lines 171 – 194
This DO loop has several things happening. It calls *Subroutine PolyDiagPnt* and places the edge vertices (their index values in the `X()` and `Y()` input arrays) into a master edge array. Then it cycles the vertices in the `tempX()` and `tempY()` arrays by 1 (e.g. vertices of order 12345 would be 23451).
6.3.4) Lines 198 – 221
Since Section 6.3.3 determined the vertices adjacent to each vertex in the convex polygon, this DO loop simply picks a starting vertex and orders the vertices in a clockwise or counter-clockwise direction (unsure which direction, but doesn't matter for what is needed).
6.3.5) Lines 223 – 224
From Section 6.3.4 the `tempX()` and `tempY()` arrays have the vertices in correct order, so these two lines transfer the data back to the `X()` and `Y()` output arrays.

6.4) Subroutine PolyCentroid
This subroutine finds a point that is within a convex polygon (a centroid). It is not used by any program, but it is kept in case it was needed and will not be broken down line-by-line.

6.5) Function WithinPoly
This function is the true workhorse of the fractional-area technique. It simply states whether a test point is within a given convex polygon or not, but the code is some of the most complex for this program. If this function could be made more efficient, it would drastically improve the speed of main program since *WithinPoly* is generally called millions or billions of times during a program run.

6.5.1) Lines 357 – 358
Allocate and initialization of temporary arrays for manipulating the polygon vertices in the function.
6.5.2) Lines 360 – 366
For function efficiency, this IF statement quickly determines if the test vertex is outside a box defined by the maximum and minimum of the X and Y polygon coordinates. If true, the test point is labeled as being not within the polygon and returned to the calling program.
6.5.3) Lines 369 – 394
Finds and removes any duplicate vertices from the polygon; identical to Sections 6.1.2, 6.1.3, and 6.1.4.
6.5.4) Line 397
Call *OrderVertices* (Section 6.3) to put the polygon vertices in the correct order.
6.5.5) Lines 400 – 569
A messy collection of nested IF statements (in a master DO loop) that answers the question of which side of the line is each point on? "The line" refers to an edge of the polygon; the DO loop cycles through all the polygon vertices. A flag variable is triggered when the test point is outside a polygon edge.
6.5.6) Lines 571 – 577
If the right amount of flags equals the number of polygon vertices, the overall function results as true (1). If not, then the function results in a false (0).

6.6) Subroutine IP_Frac_Area_Calc
This subroutine calculates the fractional area for the synthetic dataset test. A modified version of this code is used for the digital dataset.

6.6.1) Line 615
Call the subroutine *Data_Initialize* to initialize the input information in the module *Input_data*. At the time of writing, the `IP_scene` input variable is no longer required due to later code improvement; just the empty brackets now.
6.6.2) Line 617
A frequently used ratio throughout the rest of the code that only needs to be calculated the one time for efficiency purposes.
6.6.3) Lines 619 – 620
Open and start entering information into an output file (*AreaResults.txt*). All write statements referencing 60 will be written to this file and will not be mentioned in further analysis of this subroutine.
6.6.4) Lines 623 – 630
Open the file *RandomDPgrid.txt* and read in the desired amount of values. This file just contains a randomly-generated list of double-precision values within a range of 0 to 100. These values are the simulation DP values that will be solved for.
6.6.5) Lines 636 → 786
This large DO loop iterates through all IPs

6.6.6) Lines 637 – 640
Complicated array values that only need to be set once for each IP; the result is the coordinates of all four corners of a selected IP (selected by the DO loop of Section 6.6.5).
6.6.7) Lines 642 → 786
These DO loops cycle through all DPs in their index form (DP_{12}, DP_{47}, …DP_{xy}) rather than vertex form.
6.6.8) Line 647
Initialize the array `IvD_XY()` to -80 to quickly identify if and when a positive value is overwritten.
6.6.9) Lines 650 – 705
Populating the array `IvD_XY()` with the values of the vertices of the intersections between the IP and DP gridlines. Same workflow as *Subroutine IP_vs_DP_intersection* (Section 5.5).
6.6.10) Lines 707 – 712
Calculating the four corner vertices for both the current DP and IP; unsure of the reason for the IP corner vertices redundancy (compare `Xi` and `Yi` to Section 6.6.6).
6.6.11) Lines 718 – 740
This DO loop finds any DP vs IP intersection vertex that is within both the DP and IP ("within" also includes any point on the exact boundary; one `Toler` value outside the boundary to be precise). The vertices that pass this test are some of the vertices of the convex polygon whose area needs to be determined. This is the most concise and simple process that I thought of for this problem; all my other ideas were exceedingly long and next to impossible to code.
6.6.12) Lines 742 – 762
This DO loop determines which (if any) DP or IP vertices are part of the convex polygon described in the previous section (6.6.11). After this loop, the vertices for the convex polygon will be complete.
6.6.13) Lines 764 – 765
Allocate and initialize the array `Poly()` for use as the holder of the convex polygon vertices values.
6.6.14) Lines 767 – 773
Populate the array `Poly()` with values determined in Sections 6.6.11 and 6.6.12.
6.6.15) Line 775
Calculate the area of the convex polygon and normalize it with the area of the IP.
6.6.16) Line 777
For the simulation data, calculate the IP value from the corresponding DP values (set in Section 6.6.4).
6.6.17) Lines 784
In order to have Section 6.6.14 work and dynamically allocate the array size needed, this deallocate statement must be here for the next iteration of the encompassing DO loops.

6.7) Function FindDet
This function was written by Louisda16th (a.k.a. Ashwith J. Rego) to calculate the determinant of a square matrix. This function was never used in the rest of the code, but has been kept around in case it was needed.

6.8) Subroutine FINDInv
This subroutine was written by Louisda16th (a.k.a. Ashwith J. Rego) to calculate the inverse of a square matrix. This function was never used in the rest of the code, but has been kept around in case it was needed.

7) Main_Numerical_Testing.f95 and AreaManagement.f95
These files are so related and mutually dependent that I have to describe them together.

7.1) Program Main_Numerical_Testing
This is one of the main control programs for the synthetic data processing. Program name reflects the managing of the whole process, but that didn't happen according to plan. This will work for synthetic data only; see Section 10 for digital data processing, although this was the base code from which Section 10 was written. As with synthetic data processing programs, changes to the *Module Input_Data* and some subroutines may have to be tweaked to have it work.

7.1.1) Line 26
Opens an output file (`VertexOutput.txt`) to transport data from one section to another; relic from when this code was in two separate programs. This dependency could probably be removed with the appropriate modifications to the code.
7.1.2) Lines 29 → 96
Cycle through all input images
7.1.3) Line 32
Call the subroutine *Data_Initialize* (Section 4.2); in the current revision of the module *InputData* and this subroutine an argument is not needed.
7.1.4) Lines 39 – 45
Calling the subroutines *Num_Vert* and *All_Vertices* to calculate the necessary sizes to dynamically allocate the arrays used later.
7.1.5) Lines 48 –58
Populate the `temp` and `temp_type` arrays with a subset of the total vertices, specifically only the vertices within the IP grid area.
7.1.6) Lines 60 – 61
Deallocate and re-allocate the `All_XY` and `XY_type` arrays to the smaller subset described in Section 7.1.6.
7.1.7) Lines 63 – 79
Writing data to a text file opened in Line 46 (Section 7.1.2); this should be self-explanatory.
7.1.8) Lines 80 – 86
This DO loop runs through all the vertices and re-populates the arrays from Section 7.1.7. Additionally a data is output to the text file opened in Line 46 and the temporary arrays are deallocated.
7.1.9) Lines 88 – 91
Setting the outer boundaries of the DP grid index values

7.1.10) Lines 98
Writes one more line of information at the end of the entire text file (from Section 7.1.2).
7.1.11) Lines 101 – 104
Write statement formatting statements for most of the write statements previously in this program.

7.2) Program AreaManagement
This is one of the main control programs for the synthetic data processing. Program name reflects the managing of many polygon areas. This will work for synthetic data only; see Section 10 for digital data processing, although this was the base code from which Section 10 was written. As with synthetic data processing programs, changes to the *Module Input_Data* and some subroutines may have to be tweaked to have it work.

7.2.1) Line 35 – 54
Re-open the same file (`VertexOutput.txt`) as Section 7.2.2; this is relic of when this program was two separate programs. This section of code allocates arrays, populates them with information from the file, and then closes the file when it is finished. It could be removed with the right tweaking of the program.
7.2.2) Lines 56 – 59
Initiate and allocate variables and arrays.
7.2.3) Line 61
Open a data file, `MatrixInput.txt`, to record output information. I do realize that it is a terrible filename; please change.
7.2.4) Lines 65 → 176
This main DO loop cycles through all the input images.
7.2.5) Lines 70 → 113
Open, read, and then close `VertexOutput.txt` for the second time. Details following:
7.2.6) Lines 73 – 75
This DO loop simply "fast-forwards" through the file depending on which input image is current.
7.2.7) Lines 80 – 81
Allocate arrays: different for each IP scene (to not bog down memory more than necessary for large files)
7.2.8) Lines 84 – 100
Selectively extract information from `VertexOutput.txt` headers for each input image.
7.2.9) Lines 101 – 110
Extract only the IP grid intersection vertices. All other vertices are re-calculated; this is another potential efficiency improvement.
7.2.10) Lines 118 – 127
Populates an array for the organization of the IP grid intersection vertices into their respective IPs (in index notation).
7.2.11) Lines 130 – 132
Allocate and initialize arrays for the IP values and fractional-area polygons.

7.2.12) Lines 134 – 136
Call the subroutine *IP_Frac_Area_Calc* (Section 6.6). This subroutine will return with the values for the IPs and fractional-area polygons.
7.2.13) Lines 138 – 143
Write out the sum of the fractional pieces of each DP (should be 1, if not there is a problem) with a given IP and loop through all IPs
7.2.14) Lines 145 – 153
Re-use `i_IP_Inter` as an indexing array for DPs
7.2.15) Lines 155 – 171
Populate IPvector and FAmatrix for the matrix solution where [FA]*[DP]=[IP] is solved for [DP]
7.2.16) Line 175
This deallocate statement must be at the end of this DO loop in order to use the arrays again when the loop cycles
7.2.17) Lines 178 – 183
Allocating arrays to use for the matrix solver section
7.2.18) Lines 187 – 200
Excluding the write statements, this section initiates many variable and arrays used for combinations of IP equations filling a number-of-DP-sized matrix. See page 5-6 for detail of why the combinations are needed.
7.2.19) Lines 201 → 264
This DO loop will iterate until all combinations or a preset limit (Line 259) has been reached. Within each combination, the matrix is attempted to be solved. A different approach may improve the efficiency of this algorithm, since loops several million times for each matrix.
7.2.20) Lines 202 – 205
Set the subset of IP equations into their arrays. The `subset_i` array will be moved to the next combination at the end of the main DO loop (Section 7.2.19)
7.2.21) Lines 207 – 210
If the input matrix (the subset) has any column that is all zeros then flag the column. Another major improvement for this piece is not to just loop over this array (Line 212), but to make a smaller array and solve that one.
7.2.22) Lines 212 – 232
If there are no columns in the matrix with only zeroes, then the solver starts. Subroutine DGESV from LAPACK is used for the solution. If the returning answer is outside the expected value range, then it skips adding it to the output file as a valid result.
7.2.23) Lines 235 – 238
Compares the subset combination to the final combination and flags if subset has reached the final combination.
7.2.24) Lines 242 – 255
Iterate the subset to the next subset
7.2.25) Lines 257 – 263
This section of code is a counter with a stop. Line 257 controls how many combinations will be iterated until stopping (such as 1e7 = 1×10^7 times) and Line 259 controls the progress numbers that display to the screen (e.g. 1e5 combinations per number).

7.2.26) Lines 266 – 272
Some housekeeping and final write statements.

8) StatAnaylzer.f95

8.1) Program StatAnaylzer

This program is a histogram creater for the synthetic dataset results. It also should be able to be used for the digital data results as well, but those results never reached that stage of analysis, so this remained untested for that purpose.

8.1.1) Lines 10 – 14
Initializing variables for the histogram creation:

- `n_DPval` = the number of DP variables
- `n_bin_hlf` = half of the bins around a central point (the "perfect" answer). Therefore the histogram ranged from `-n_bin_hlf` to `+n_bin_hlf`
- `data_strt` = Line number where the data starts in the matrix results file.
- `data_end` = Line number where the data ends in the matrix results file.
- `Toler` = the bin size

8.1.2) Line 16
Allocation of arrays based on the initialized variables
8.1.3) Lines 18 – 22
Read in the "perfect" synthetic DP values. This step would be omitted with real data (where the answer is not known previously), unless a higher resolution dataset already existed and comparison was desired.
8.1.4) Lines 24 – 26
Initialize arrays
8.1.5) Lines 28 – 29
Open the matrix results file and read through (past the header results) till the start of the solutions.
8.1.6) Lines 31 → 74
Cycle through all matrix solutions
8.1.7) Line 32
Read in one solution line (with `n_DPval` DPs space delimited)
8.1.8) Lines 33 → 73
Cycle through each DP and compare with "perfect" DP value
8.1.9) Lines 34 → 52
Determines if the sample DP value is on the positive side of or equal to the "perfect" DP value
8.1.10) Line 35
This line will determine the largest values in the dataset for each DP
8.1.11) Lines 36 → 51
Cycles through each positive bin and determine if the sample DP value fits within it

8.1.12) Lines 37 – 41
If the sample DP value is within ½ `Toler` of the "perfect" value it is recorded as such. Line 39's final value will be combined with Line 59's final value to create a single bin of `±0.5*Toler` from the "perfect" DP value
8.1.13) Lines 42 – 44
If the sample DP value is outside the bins, the last bin is a catchall for all positive outliers
8.1.14) Lines 45 – 50
Determine into which bin a sample DP value fits
8.1.15) Lines 54 → 72
Determines if the sample DP value is on the negative side of the "perfect" DP value
8.1.16) Line 55
This line will determine the smallest values in the dataset for each DP
8.1.17) Lines 56 → 71
Cycles through each negative bin and determine if the sample DP value fits within it
8.1.18) Lines 57 – 61
If the sample DP value is within ½ `Toler` of the "perfect" value it is recorded as such. Line 59's final value will be combined with Line 39's final value to create a single bin of `±0.5*Toler` from the "perfect" DP value
8.1.19) Lines 62 – 64
If the sample DP value is outside the bins, the last bin is a catchall for all negative outliers
8.1.20) Lines 65 – 70
Determine into which bin a sample DP value fits
8.1.21) Lines 77 – 82
Open a file to store the histogram and writes the header of the file
8.1.22) Lines 83 – 89
More header information: lists out a distribution analysis bound by the defined number of bins.
8.1.23) Lines 91 – 93
List out each bin number and how many sample DP values fell within it and writes this out for all DP values on a space-delimited line
8.1.24) Lines 96 – 98
Text formatting lines used in Sections 8.1.22 and 8.1.23.

9) BPP PreProcessor.f95

9.1) Program BMP_PreProcessor

This program is designed to pre-process the digital data bmp files to a usable input for *Program RealDataMainProgram*. Due to the length of this program, only the most crucial and/or potentially confusing sections will be described. It is assumed that write statements, variable declarations, and array allocations are straight-forward to understand to the reader.

9.1.1) Lines 14 – 16
These processor-dependent parameters are for different-sized integer arrays. The arrays can be set for storing 1, 2, and 4 byte integers. This line is needed here since the parameter will be different across different hardware setups.
9.1.2) Lines 60 – 67
Input images' properties needed to calculate all the vertices. See the comments next to each variable in the declaration section (Lines 23-43) for details.
9.1.3) Lines 73 → 95
Cycle through all the input images
9.1.4) Lines 75 – 78
Initialize the array IP_c_limit to the first images' bottom-left corner. This array will be used in Section 9.1.6 to hold the maximum boundaries of all images combined.
9.1.5) Lines 81 – 89
Calculate the four corners of an input image (assumed to be parallelogram shaped)
9.1.6) Lines 91 – 94
Tests the corners to the maximum or minimum values in the X and Y directions; if a corner exceeds a comparative value, then it becomes the new maximum/minimum value.
9.1.7) Lines 101 – 102
The array `DP_limit` (the maximum limits of a rectangle of DPs)
9.1.8) Lines 113 → 256
Cycle through all input images
9.1.9) Lines 118 – 126
Calculate the IP grid corners; identical to Section 9.1.5
9.1.10) Lines 129 – 131
Determines the best DP grid rectangle that encompass the entire IP grid
9.1.11) Lines 138 – 144
Cycle through all DP vertices and determine if each is within the IP grid area
9.1.12) Lines 152 – 158
Determines if any part of the DP is within the IP grid area; this refines the selection of DPs from Section 9.1.10
9.1.13) Lines 168 – 175
Calculates the intersections of the IP grid and stores the vertices in the array `IP`; functionally the same code as Section 5.4
9.1.14) Lines 185 – 222
Calculates the intersections of the IP and DP grids and stores the array; functionally the same code as Section 5.5 (see for additional details).
9.1.15) Lines 231 – 250
Determines which IP/DP intersection vertices are within the IP grid area and flag the difference
9.1.16) Lines 262 – 281
BMP file header values assigned to variables. See the comments at the end of lines for more information regarding the variable on that line
9.1.17) Lines 283 – 308
Creating a Windows 24-bit BMP file showing the coverage of the DP pixels and the overlapping relationship of the input images. See Figure 10 to see the resulting BMP.

9.1.18) <u>Lines 314 – 316</u>
Opens three output files that will be binary data files to conserve space (as opposed to ASCII data files which can be opened in a word processor and is human readable)
9.1.19) <u>Lines 320 – 323</u>
Write all information needed for the DP grid and all are within the IP grids
9.1.20) <u>Lines 331 – 337</u>
Write all information needed for the IP grids
9.1.21) <u>Lines 347 – 356</u>
Creating an integer array of IP/DP intersection vertices which failed to be within the IP grids
9.1.22) <u>Lines 364 – 375</u>
Reusing the `IP` array, store only the vertices which are within the IP grids
9.1.23) <u>Lines 384 – 390</u>
Write all information needed for the IP/DP intersection vertices

10) RealDataMainProg.f95

10.1) Program RealDataMain

This program is the true workhorse of processing digital data to a potential conclusion. It will repeat much of the same code as previous programs and subroutines and, where appropriate, the reader will be referred back to previous sections for more detail. Due to the length of this program, only the most crucial and/or potentially confusing sections will be described. It is assumed that write statements, variable declarations, and array allocations are straight-forward to understand to the reader.

10.1.1) <u>Line 71</u>
Initialize the input data in the *module Input_data* (Section 4). This format was chosen to have just one easy to see file to edit preferences and data inputs, rather than scattered around the code.
10.1.2) <u>Lines 76 – 82</u>
Read in the valid DP values from `DP.vrt` (created in Section 9.1.19)
10.1.3) <u>Lines 84 – 93</u>
Read in the valid IP values from `IP.vrt` (created in Section 9.1.20)
10.1.4) <u>Lines 106 – 107</u>
Calculates two values that will be used throughout the rest of the program

- `rim` = rim around the `DP_GS`-sized square which can be include in the matrix calculation if any part of an IP falls within any part of the square
- `cntIPall` = an deliberate over-estimate of the number of IPs within the sample DP square

10.1.5) <u>Lines 110/111 → 463</u>
Cycle through all the DPs in index form (pixel 1, pixel 2) in increments of `DPsq - rim` as opposed to vertices; this is the main DO loop of the program

10.1.6) Lines 120 – 127
Efficiency IF statement: if there are no DP points or are outside the loop bounds, the specified main loop cycles (No point in trying to solve what is not possible) It is a very significant speed increase on the edges of the DP area.
10.1.7) Lines 142 → 254
Cycles through all the input images
10.1.8) Lines 148 – 162
Determine which IP vertices and how many IP vertices are within the sample DP area
10.1.9) Line 169
If no IP vertices were found for the current input image, then cycle to the next for efficiency.
10.1.10) Lines 175 – 203
Of the IP vertices within the sample DP area, which correspond with a complete IP within the sample DP area boundary conditions.
10.1.11) Line 209
If there are no complete IP, cycle to the next input image
10.1.12) Lines 214 – 232
Now that the complete IPs have been identified, determine which DPs are not covered using the *subroutine WithinPoly*.
10.1.13) Lines 242 – 248
Flag the results from the previous step.
10.1.14) Lines 256 – 265
Count the number of DPs that cover the IPs for the sample DP area
10.1.15) Lines 268 – 274
This is just counting how many more IPs than DPs and recording that to a file to be examined later. Not crucial to the operation of the program and is relatively computational minimal.
10.1.16) Lines 276 – 279
If there are more DPs than IPs in the sample DP area, the matrix solution is impossible. Therefore this cycles the `MainY` loop for efficiency. The rest of the code would still work if this wasn't here, but it is doing unnecessary work.
10.1.17) Lines 287 → 342
Cycle through all IPs within the sample DP area
10.1.18) Lines 289 – 290
Call the *subroutine Frac_Area_Calc* attached at the end of the program. It has been slightly modified from Section 6.6 subroutine, but the same logic is used.
10.1.19) Line 293
Calculate a padding variable for reading the input data from BMP form
10.1.20) Lines 296 → 341
Cycle through all IPs → Note: this is NOT restricted by the sample DP area!
10.1.21) Line 297
Cycle the DO loop L2 (Section 10.1.20) if the IP is not within the sample DP area.

10.1.22) Line 299
For every IP within the sample DP area, increment by one. This is used while an intermediate number and will be compared to the previously calculated number of IPs after the Section 10.1.17 loop is complete.
10.1.23) Lines 301 – 320
Cycle through all DPs and, after a number of error checks, populate the array that symbolized the square matrix to solve.
10.1.24) Lines 322 – 340
Read the selected IP's values from the input image (24-bit BMP format) and this creates the IP vector part of the matrix equation we are trying to solve.
10.1.25) Lines 344 – 354
Convert the input data into 4-byte integers and into human readable numeric format (a quirk of the integer storage as a byte per color)
10.1.26) Lines 359 –362
A redundant error check: If the two calculated number of IPs do not match there is a bug somewhere.
10.1.27) Lines 369 → 453
Cycle through all three color components of the input images (blue, green, and red) and solve the matrix separate for each since while the geometry has not changed the IP value will.
10.1.28) Lines 370 – 446
Cycle through IP equation combinations and solve the matrix; see Sections 7.2.18 – 7.2.25 for details. The code is identical with the exception of Lines 405–407 where the separate colors are accounted for.
10.1.29) Lines 449 – 452
If no solution is obtained from one of the colors, cycle the sample DP area (the `MainY` DO loop). The reasoning behind this: a resulting picture cannot be created if one of the colors has no value.
10.1.30) Line 462
Like most of the deallocate statements, this deallocate statement must be immediately before the "End DO" statements for the `MainY` and `MainX` DO loops.

10.2) Function PolyArea
This function is identical to the code described in Section 6.1

10.3) Subroutine PolyDiagPnt
This subroutine is identical to the code described in Section 6.2

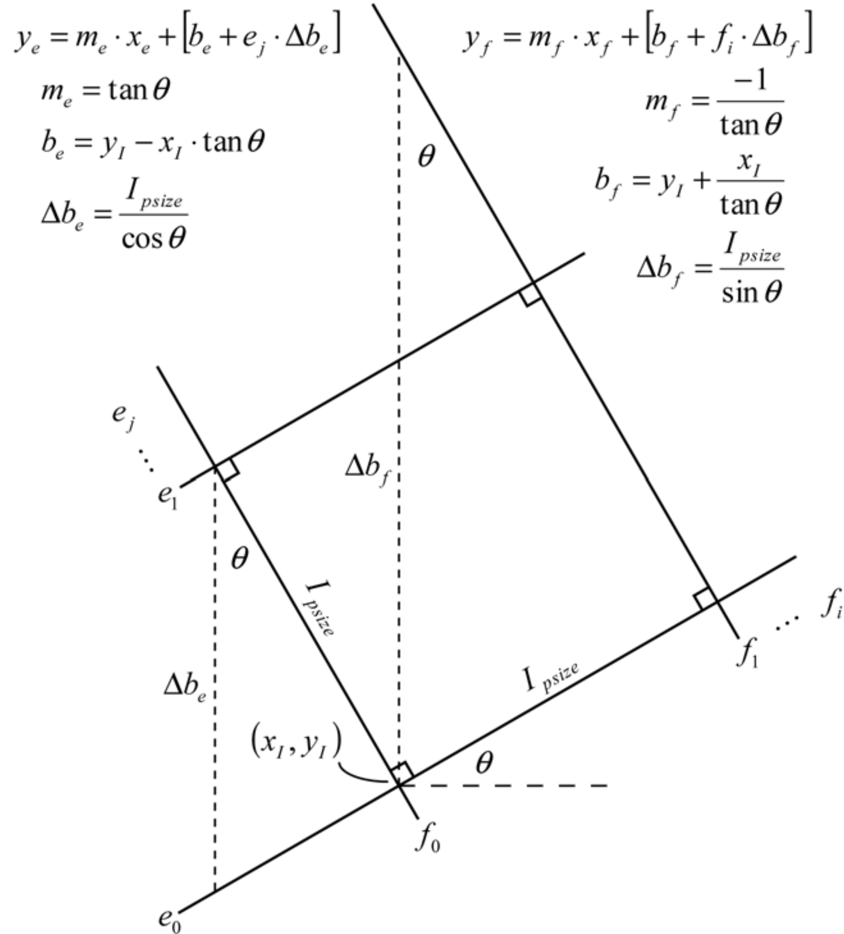
10.4) Subroutine OrderVertices
This subroutine is identical to the code described in Section 6.3

10.5) Function WithinPoly
This function is identical to the code described in Section 6.5

10.6) Subroutine Frac_Area_Calc

This subroutine is similar to the code described in Section 6.6. The modifications made were only to get it to work with digital data and the logic remains identical.

Appendix B: IP Pixel Grid Vertices Calculation Explanation



Other definitions:

$I_{px} \Rightarrow$ number of initial pixels in the x direction

$$f_i = f_{i-1} + 1 \text{, where } f_0 = 0,\ f_i \text{ is an integer, and } f_i \le I_{px} \tag{1}$$

$I_{py} \Rightarrow$ number of initial pixels in the y direction

$$e_j = e_{j-1} + 1 \text{, where } \mathrm{e}_0 = 0,\ \mathrm{e}_j \text{ is an integer, and } \mathrm{e}_j \le I_{py} \tag{2}$$

For e lines:

$$y_e = \tan\theta \cdot x_e + \left[y_I - x_I \cdot \tan\theta + e_j \cdot \frac{I_{psize}}{\cos\theta} \right]$$

$$\Rightarrow y_e = \tan\theta \cdot (x_e - x_I) + y_I + e_j \cdot \frac{I_{psize}}{\cos\theta} \tag{3}$$

$$\Rightarrow x_e = x_I + \frac{y_e - y_I - e_j \cdot \frac{I_{psize}}{\cos\theta}}{\tan\theta} \tag{4}$$

For *f* lines:

$$y_f = \frac{-1}{\tan\theta} \cdot x_f + \left[y_I + \frac{x_I}{\tan\theta} + f_i \cdot \frac{I_{psize}}{\sin\theta} \right]$$

$$\Rightarrow y_f = y_I + \frac{x_I - x_f + f_i \cdot \frac{I_{psize}}{\cos\theta}}{\tan\theta} \tag{5}$$

$$\Rightarrow x_e = \tan\theta \cdot (y_I - y_f) + x_I + f_i \cdot \frac{I_{psize}}{\cos\theta} \tag{6}$$

Intersecting the *e* and *f* lines:

$$y = y_I + \frac{\frac{I_{psize}}{\cos\theta} \cdot (f_i \cdot \tan\theta + e_j)}{\tan^2\theta + 1} \tag{7}$$

$$x = x_I + \frac{\frac{I_{psize}}{\cos\theta} \cdot (f_i - \tan\theta \cdot e_j)}{\tan^2\theta + 1} \tag{8}$$