IMAGE REGISTRATION IN AIRBORNE REMOTE SENSING

by

Michael Williams Funk

A thesis submitted to the faculty of
The University of North Carolina at Charlotte
in partial fulfillment of the requirements
for the degree of Master of Science in the
Department of Computer Science

Charlotte

2006

Approved by:

Dr. Kalpathi R. Subramanian

Dr. M. Taghi Mostafavi

Dr. Kayvan Najarian

ABSTRACT

MICHAEL WILLIAMS FUNK. Image registration in airborne remote sensing. (Under the direction of DR. KALPATHI R. SUBRAMANIAN)

The primary topic is an algorithm for image registration which is especially well-suited to the types of images found in airborne remote sensing applications. A brief overview of the field of remote sensing is given, prior to discussion of the registration algorithm itself.

Image registration is the process of associating the contents of one image with those of another. For example, an aircraft might image the same terrain multiple times on different days and from slightly different perspectives. If a registration mechanism is available to warp the images' contents into a common geometric frame of reference, then a number of very useful applications become possible. These applications include three-dimensional terrain modelling, automated mapping, and monitoring environmental changes over time.

The applications of registered imagery are described, but the main focus is on the actual image registration algorithm. The algorithm is a two-stage process. The first stage, initial alignment, generates a rough estimate of the images' geometric relationship using either user input or data captured from the aircraft's navigation system. The second stage, correlated alignment, fine tunes this estimation using pattern matching algorithms on the images' pixel data. The effects of the algorithm on a number of image pairs are demonstrated, as well as several techniques for evaluating the performance of the algorithm.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: REMOTE SENSING

In a very broad sense, remote sensing can be described as acquiring information about an object or a region from a distance, by examining radiation reflected or emitted by it. In practice, remote sensing is primarily applications of airborne or satellite-based imaging systems. An imaging system can be defined as the combination of one or more sensors (for example, an infrared camera), an imaging platform (such as an aircraft or a satellite), and any supporting mechanical, electrical, or computer systems which assist in mission planning and execution, and data capture and exploitation. Supporting systems might include an image processing workstation, a *GPS* (global positioning system) receiver, an *INS* (inertial navigation system), a gimbal (a turret-like pointing system for one or more sensors), and optics to enhance the resolution of a camera, among many other possibilities.

Traditionally, images were captured on film and analyzed by hand using specialized equipment. Digital imaging devices and software-based analysis techniques have since become the norm for most applications. In some cases, this has simply streamlined the processes used by image analysts. However, it has also enabled a number of automated image processing applications which were not possible before.

The ability to perform pixel-level (or subpixel) image registration is a core requirement of many different remote sensing applications. At a very high level, image registration is the process of determining the location of common features in a pair of images of the same scene. If a particular object (such as a structure or a natural feature) is visible in two different images of the same scene, it is almost always in different pixel locations in the two images. Say that an object is visible in both images, and is identified as being at pixel coordinate (x_0, y_0) in the first image. In the second image, the same object is identified as being at pixel coordinate (x_1, y_1). The difference between the two coordinates may be due to a number of things. The object may have moved between the times at which the two images were taken. It is much more likely, however, that most or all of the differences between the two sets of pixel coordinates are due to the camera having a different position or orientation. Registration reveals the mapping between each pixel in one image and the corresponding pixel, based on image content, in the second image.

1.1 Sensors

For the applications described here, the sensors involved are generally digital imaging systems. These sensors can be loosely categorized based on a few general characteristics. The first characteristic is the frequency (or frequencies) of radiation they are sensitive to. If the sensor is sensitive to more than one frequency, it is known as a multispectral system. For example, one of the primary sensors on the LANDSAT 7 satellite is the ETM+ (enhanced thematic mapper plus). The ETM+ is a multispectral system which is sensitive to radiation centered around 8 different frequencies. From an image processing perspective, there are 8 different samples associated with each pixel; traditional image processing usually only deals with a single sample (for grayscale images) or 3 samples (for color images).

Beyond multispectral systems are hyperspectral systems. The distinction between multispectral and hyperspectral is loose and not formally defined. In general, a multispectral system samples a small number of frequencies over relatively large bandwidths, whereas a hyperspectral system samples a large number of frequencies over relatively small bandwidths.

The frequency (or frequencies) a sensor is sensitive to dictates a number of other characteristics of the system, including the nature of the radiation detected by the system and the type of information contained in its images:

- *Infrared* sensors generally detect radiation emitted by objects in a scene; the magnitude of the radiation is highly correlated with the temperature of a given object.

- *Optical* sensors detect radiation at or near the frequencies sensed by the human eye (visible light). The radiation they detect generally originates from the sun or artificial lighting, and reflects off of the objects in a scene before making its way to the sensor. The images closely resemble what a human observer would see, increasing interpretability by human analysts.

- *Microwave* sensors, or imaging radars, emit microwave radiation and detect the reflected signal. The most advanced type of imaging radar is a *SAR* (synthetic aperture radar). A SAR emits a large number of microwave pulses over a short period of time, then integrates their returns to form a single, high-resolution image. Because a SAR emits the radiation which it later detects after reflection, it is known as an *active* sensor. In contrast, infrared and optical sensors are *passive* sensors, because the radiation they detect originates from elsewhere. The magnitude of a given pixel in a SAR image is generally due to material properties (especially roughness) of an object's surface. Jakowatz [5] is an excellent overview of the SAR image formation process, as well as image processing algorithms which are unique to SAR.

The frequency also determines a number of other operational capabilities. Infrared and microwave sensors operate equally well at night, whereas optical sensors are sorely limited without daylight. Microwave radiation is not affected by smoke or clouds, so SARs are useful in many situations where infrared or optical sensors are unuseable. Unfortunately, the nature of microwave radiation is such that SAR images contain less detail and more noise than infrared or optical images. The point is, each type of sensor has its own strengths and weaknesses; there are also many applications which are unique to each one.

1.2 Supporting technologies

Many airborne imaging applications are dependent on technologies other than the sensors themselves. An essential component in most systems is a *GPS* (global positioning system) receiver integrated with an *INS* (inertial navigation system).

The GPS is composed of a network of satellites and a number of ground control stations spread throughout the world. The system is under the control of the United States Department of Defense, although it is widely used for both civilian and military applications. The purpose of a GPS receiver is to determine the user's position based on analysis of signals transmitted by the satellites. Each satellite emits a unique radio signal on two different frequencies, known as L1 (at 1575.42 MHz) and L2 (at 1227.60 MHz). The L1 signal is intended for civilian use. The L2 signal is intended for military use only. L1 is used to broadcast both a cleartext and an encrypted signal, whereas the signal on L2 contains only encrypted data. Civilian GPS receivers can access the unencrypted L1 signal; military GPS receivers with the proper cryptographic hardware and private keys can access the encrypted signals on L1 and L2. The primary benefit of a military receiver is the ability to compute a much more accurate position than is possible with the unencrypted signal alone.

Kaplan [6] provides an excellent overview of GPS technologies and applications. For greater depth, the "bible" of GPS engineering is generally considered to be the two-volume set published by the AIAA [10, 11].

The basic GPS concept is fairly simple. Every element of the GPS (satellites, ground control stations, and receivers) has a clock that is kept synchronized with every other element. The receiver maintains a continually updated database of each satellite's orbital parameters, and can compute each satellite's position in 3D space as a function of time. Signal processing hardware and software within the receiver enables the receiver to measure its range to each satellite. Therefore, the receiver knows when it received a signal, it knows which satellite a signal comes from, and it knows its distance to that satellite. Because the signal propagates through free space at the speed of light, the receiver can also compute the time of transmission of the signal. Using its knowledge of the satellite orbits, it computes the position of that satellite at the time of transmission.

At this point, the receiver knows its range to a number of satellites, and it knows each satellite's position in 3D space. Conceptually, one could draw a sphere around each satellite, in which the radius of the sphere is equal to the range to the receiver. The receiver must be located at some point on the surface of that sphere. Multiple satellites create multiple spheres. Ideally, all of the spheres would intersect at a single position. By determining that point of intersection, the receiver determines its own position. Unfortunately, errors in measurement ensure that there is never a single, perfect point of intersection. The receiver gets around this by reinterpreting the problem as a (possibly overdetermined) linear system (assuming at least four satellites are in view), and generating a least-squares solution for the receiver's most likely position. GPS receivers typically generate solutions at 1 Hz.

Another important piece of hardware is an inertial navigation system. An INS is a device capable of measuring its own acceleration in six degrees of freedom: translation and rotation about three perpendicular axes. Accelerometers measure changes in position, and gyros measure rotation. If an INS has a known starting position and orientation, then differential equations can be used to to derive an updated position and orientation at any point later in time. A typical INS might generate updated position/orientation solutions at 100 Hz or more.

The combination of a GPS receiver and an INS is much greater than the sum of the parts. The two technologies have very complementary error characteristics. GPS receivers are less accurate, but the accuracy does not degrade over time. If anything, the solution tends to get more accurate. INS measurements are extremely accurate, but have a small bias that accumulates over time as each INS sample is integrated to create a new solution. INS errors will continue to grow without bound, but the magnitude of GPS errors stay relatively constant. INS errors can be characterized as a constantly increasing bias with very little variance, but GPS errors are much more random and have very little bias.

A Kalman filter is a very effective means of integrating a time series of noisy measurements from multiple sources, and produces a single filtered measurement which minimizes the errors in the system. Kalman filters have been especially useful in the implementation of integrated GPS/INS navigation systems. The definitive work on Kalman filtering for GPS/INS systems is [4]. Kalman-filtered GPS/INS systems are widely available as *COTS* (commercial, off-the-shelf) components for air, land, and sea-based navigation systems. Such systems are also the key component in modern precision strike munitions, including "smart bombs" such as the JDAM (Joint Direct Attack Munition).

Practically speaking, a Kalman-filtered GPS/INS provides a highly accurate measurement of an aircraft's position and velocity in 3D space as well as its orientation (roll, pitch, and yaw), at 10 Hz or more. By integrating the GPS/INS with the imaging system, one can associate each image with a position and orientation of the platform. If the range to the ground is known (for example, using a laser rangefinder or through computations based on a topographic terrain model), then the 3D position of each pixel in each image may be estimated as well. Obviously this is of great assistance in applications such as mapping or weapon targeting, but it also opens up a wide range of applications which require access to large databases of geolocated imagery.

GPS/INS systems are widely available as off-the-shelf components, in very compact form factors. The C-MIGITS III is a typical integrated GPS/INS. Figure 1.1 is a photograph of the unit. It is extremely lightweight, and is approximately 10 cm tall. Interfacing to the C-MIGITS is simply a matter of sending commands and receiving updates via an RS-232 port.

Another technology which is significant to airborne imaging are gimbal systems. A gimbal provides the ultimate packaging for airborne sensors. A gimbal is a stabilized turret containing one or more cameras. The most advanced gimbals contain sophisticated optics which are shared between all available cameras, an integrated GPS/INS and laser rangefinder for image geolocation, and advanced control systems which allow an operator to move the gimbal with a joystick or instruct it to point at a particular position in 3D space. Without a gimbal, cameras are generally mounted on the side of the fuselage or inside the aircraft itself, peering through a hole cut in the floor of the cabin.

1.3 Applications

The most common applications of remote sensing are related to mapping and intelligence gathering. Civilian applications might be concerned with environmental monitoring or mapping large regions of wilderness. Military applications tend to fall under the category of *ISR* (intelligence, surveillance, and reconnaissance) missions. Sophisticated remote sensing systems which are highly integrated with wartime activities and decision-making are referred to as *C4ISR* (command, control, communications, computers, and ISR) systems. C4ISR systems typically involve gimballed imaging systems on *UAV*s (unmanned aerial vehicles) with sophisticated realtime communications and control systems, capable of delivering up-to-the-minute intelligence to battlefield commanders in immediately useful formats. The C4ISR community is large and active and has a number of journals and conferences devoted to it.

The following passage from United States Department of Defense's 1996 Annual Defense Report demonstrates the importance of C4ISR to the US military's long-term planning and evolution:

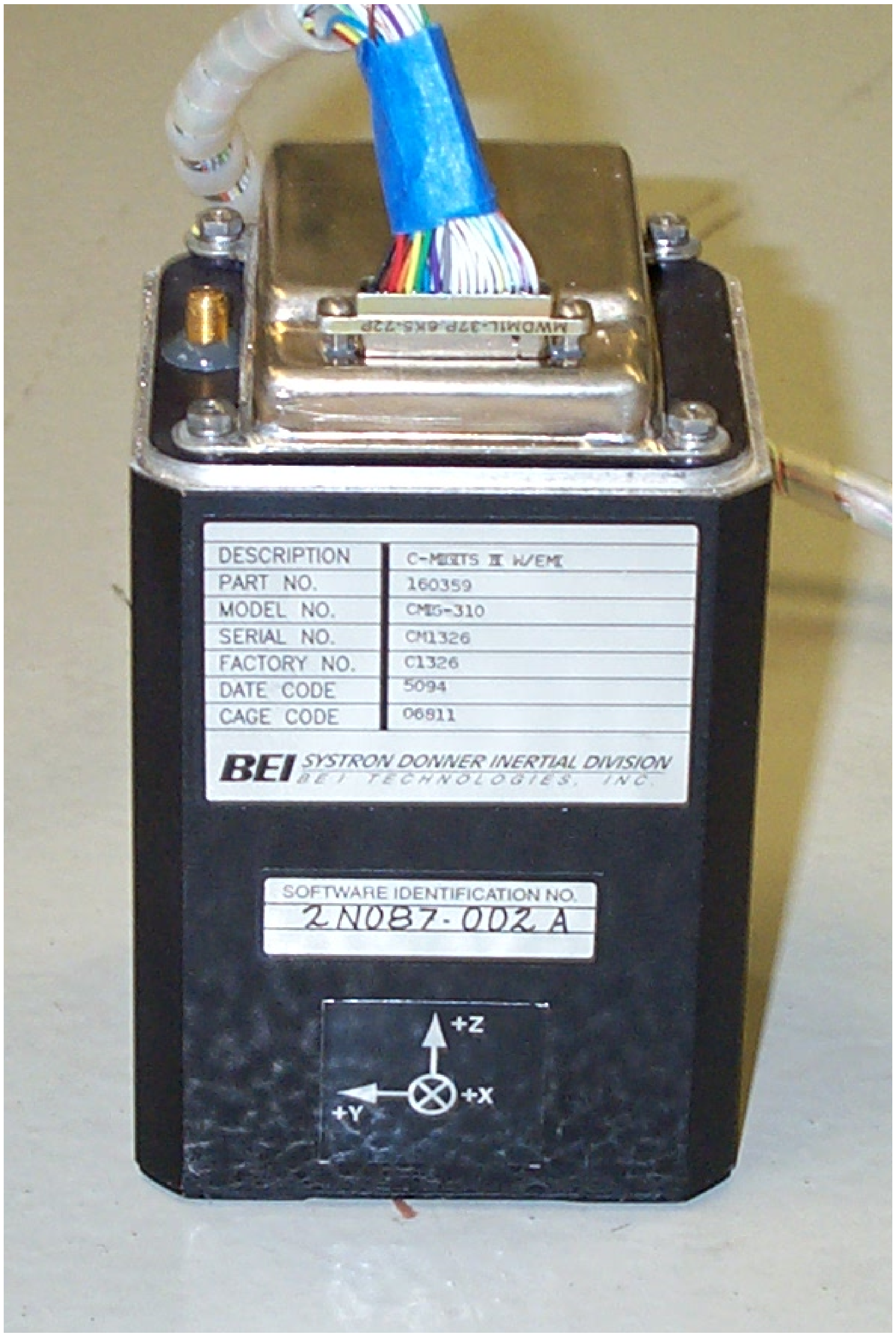


FIGURE 1.1: The C-MIGITS III GPS/INS navigation system.

> The focus of surveillance and reconnaissance is directly supporting warfighter dominance of the battlefield. Battlefield dominance requires: (1) battlespace awareness to provide warfighters with better, missionfocused and tailored understanding of all force dispositions, capabilities, and intentions; (2) an advanced C4ISR infrastructure to disseminate battlespace awareness information rapidly; and (3) precise targeting information for precision guided weapons, and other lethal and non-lethal offensive systems. Improved intelligence, reconnaissance, and surveillance provides the tools to counter the fog of war, and to enable operations to take place within the opponent's decision cycle time. Thus, United States forces can take and hold the initiative, increase operational tempo, and concentrate power at times and places of their choosing.

Some of the most interesting remote sensing applications are in the field of *photogrammetry*. Photogrammetry is the development and application of analysis techniques on images, to make measurements of features present in an image. This is a very broad definition for a very broad field. Common photogrammetric applications include the generation of topographic maps of a region from two or more airborne or satellite images, or measuring the dimensions of a large structure or natural feature. Photogrammetry predates digital imaging technology by a century or more; the original photogrammetric techniques were developed using image prints on developed film. The introduction of digital imaging has allowed the field to flourish in directions undreamt of by its pioneers.

Another interesting application of remote sensing is multispectral image analysis. A number of satellites provide high-quality multispectral imagery of large regions of the Earth's surface; the images produced by these satellites contain many (six or more) samples per pixel, taken from different frequencies. Multispectral analysis involves using classification algorithms to identify characteristics of the region that was imaged. Certain combinations of pixel values might indicate different types of soil, the presence of a certain type of natural resource, or the degree of human encroachment on an area. In a sense this is just making assumptions about things based on color, but the range of bands available to a multispectral sensor provide much more information than what can be conveyed by visible light. An excellent introduction to multispectral imaging systems (and remote sensing in general) can be found in [13].

CHAPTER 2: IMAGE REGISTRATION CONCEPTS

Image registration can be defined as an algorithm which maps the pixel space of one image into that of another. This mapping is based on the presence of common elements in both images. The two images may differ for any number of reasons; most commonly, the same scene was imaged by cameras having different positions and orientations. The images may also have been taken at different times, resulting in changes in the scene's illumination or changes in the scene itself. Most likely, all of these differences are present in varying degrees.

One important special case is when the images were generated by different imaging systems, for example an infrared camera and a SAR. This is known as multimodal or multispectral imaging. Different frequencies of radiation provide very different types of information about the object being imaged. What determines the relative lightness or darkness of a given object in an image varies with the frequencies a camera is sensitive to. A hot spring might be the most prominent feature in an airborne IR image, but would not even be visible in a SAR image. This is an extreme example, however. In general, many basic structures will be visible in both images, but the pixel intensities and textures which highlight those structures will be very different.

The registration algorithm presented here is effective for overcoming differences in camera geometry, changes in a scene over time, and minor differences in illumination patterns. However, it is not effective for multispectral cases. This is because it is based on pattern-matching through texture analysis; it assumes that similar patterns of pixel intensities surround matching areas and objects in both images. It could probably be modified to work for certain multispectral cases by filtering the input images prior to registration. For example, passing an image through a high-pass filter highlights boundaries and diminishes regions having smooth gradients or relatively constant pixel values. This would enhance the effect of shape on the pattern-matching functions, while decreasing the effect of shading.

In image registration, one of the input images is referred to as the base image, and the other is called the warp image. The process of registration will result in the generation of one or more image transforms which will be applied to the warp image; the base image is unchanged. After registration, an object visible in both images will be present at the same pixel coordinates in both images. For example, prior to registration an object is visible in the base image, centered at the pixel coordinate (x, y). The same object is visible in the warp image at the pixel coordinate (x', y'), which is most likely not equal to (x, y). After registration and warping, x' will equal x (and y' will equal y) for all objects common to both images.

Figure 2.1 is an example of a base and warp image pair, prior to image registration. Figure 2.2 shows the results of sucessful image registration. The bottom image has been warped into the pixel space of the top image. Nonoverlapping regions of the warp image are given black pixels, to indicate that the original image did not contain information about that region. The base image (the image on top) was not modified during registration.

2.1 Image registration applications

There are a number of interesting applications that require pixel-level image registration. Some of the most common are terrain modeling, change detection, and mosaicking.

2.1.1 Terrain modeling

Terrain modeling is the process of generating topographic maps of a region from sets of images with different camera geometries. By imaging a region from different positions at different angles, distortions are introduced in the images which are the direct result of the scene's topography. By registering the images together, one determines the correspondence between each point in one image to another point in

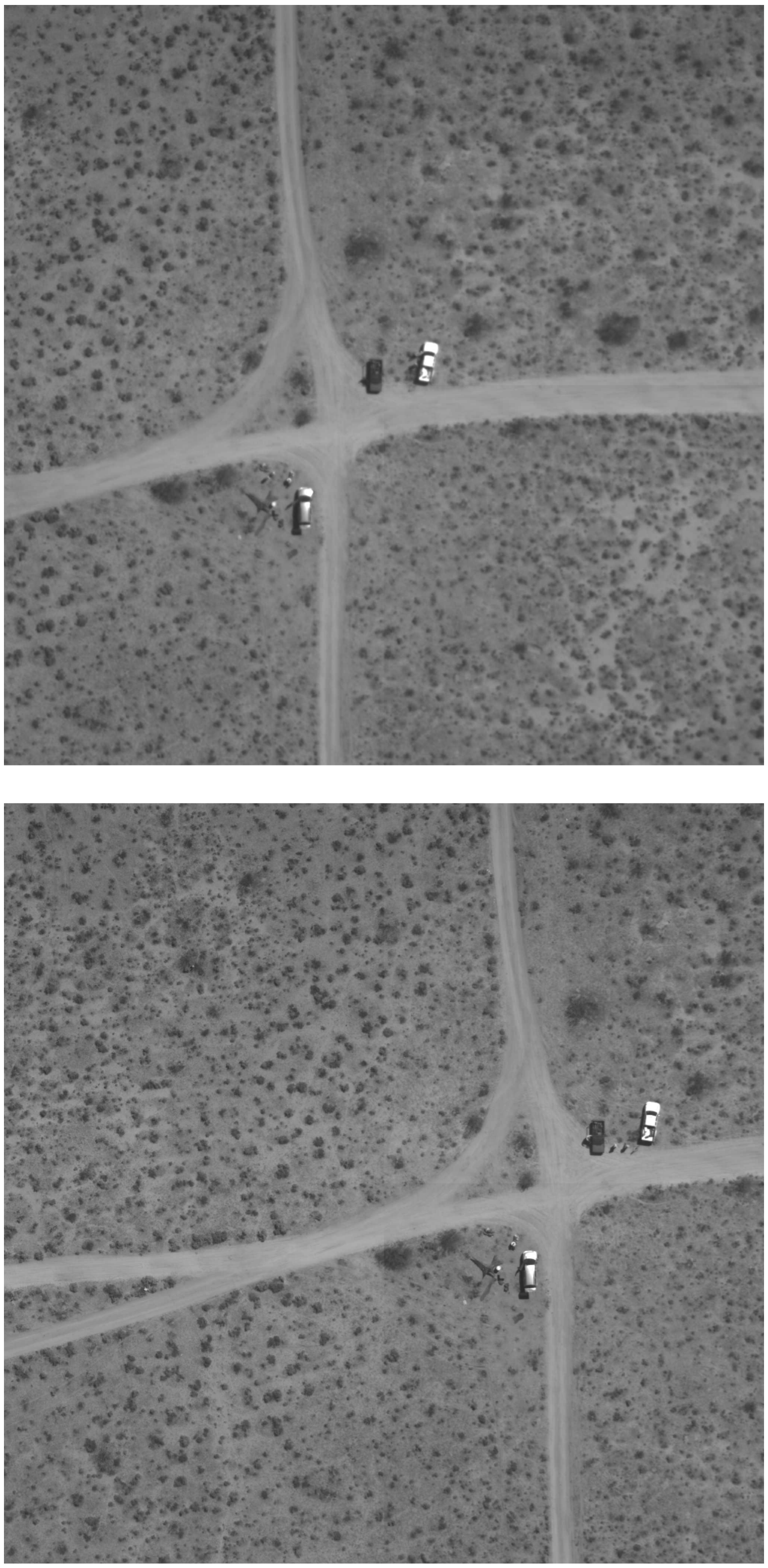


FIGURE 2.1: An image pair prior to registration.

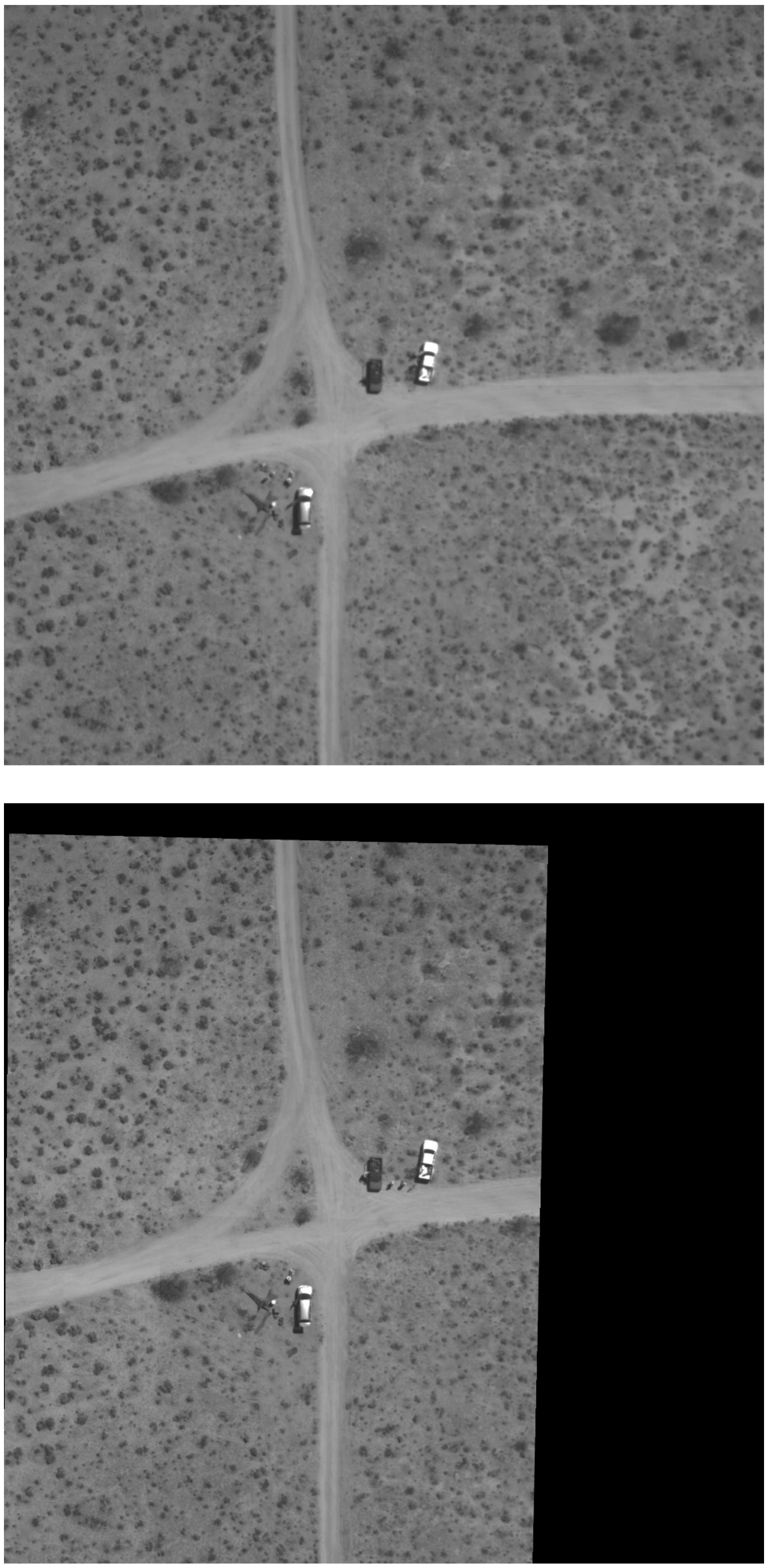


FIGURE 2.2: The image pair after registration.

another image. Using what is known about the camera's position and orientation at the time each image was taken (usually from GPS/INS nav data), it is possible to triangulate 3D positions for each pair of corresponding points. This results in a 3D point cloud, in which each point has a color associated with it as well as a position. Using an advanced interpolation technique such as Delaunay triangulation, one can resample the z-axis values onto a regularly spaced xy grid, resulting in a *DEM* (digital elevation model) of the scene. A DEM is essentially a discrete sampling of the terrain's topography.

An additional data product that can be produced by this process is an *orthorectified image*, sometimes called an "ortho" for short. An ortho is an airborne or satellite image in which all effects of perspective have been removed; the image resembles an orthographic projection of the terrain onto a plane that is tangent to the surface of the Earth. Recall that prior to the generation of the DEM, we had a 3D point cloud in which each point had an associated color (taken from one or both of the source pixels in the input images). Creating an ortho from this point cloud is identical to creating a DEM, except we are resampling the pixel colors to the uniform xy plane rather than the z values.

2.1.2 Change detection

While terrain modeling requires two or more images of the same scene from different perspectives, change detection works best when the input images are from very similar perspectives. Terrain modeling works by examining the differences in a pair of images resulting from changing the camera's position in space; change detection is concerned with examining the differences in a scene over time. Two or more images are taken of a scene at different times; the time interval is a function of the types of changes one is trying to detect. If one is interested in measuring the motion of glaciers, the images may be separated by years. If one is interested in measuring the effects of a missile strike, the images may be separated by seconds. In either case, the camera should have the same position and orientation in each image, or as close as can be managed. The images are then registered and subtracted. The result of the image subtraction is referred to as a *difference image*, because it identifies things which have changed over the time interval between the two images. In military applications, a difference image might highlight tire tracks or disturbed soil where troop movements had occurred between the times the two images were taken.

2.1.3 Mosaicking

Mosaicking is the process of combining images of overlapping regions into a single large image mosaic. The regions depicted in the images must overlap, so that image registration algorithms can align them to each other on a pixel-level basis. This technique is also useful for aligning arrays of cameras. Some imaging systems are designed so that multiple cameras with overlapping fields of view will acquire images simultaneously. Postprocessing software then combines the images into a single large image, simulating a single camera with a much larger field of view. This requires knowing each camera's position and orientation relative to every other camera. This can be physically measured, however in airborne applications the cameras can easily become misaligned due to vibration. By registering the overlapping regions of the cameras' images to one another, the precise camera calibration parameters can be directly measured. This is then fed back in to the postprocessing software, which will use this information to better integrate the images into larger, composite images.

2.2 Cross correlation

Cross correlation is a central function in the image registration algorithm presented here. The amount of time it takes to register a pair of images is almost entirely dominated by the efficiency of the implementation of the cross-correlation function.

The most common description of the cross correlation of two images is as follows:

$$f(x,y) \circ g(x,y) = \frac{1}{MN} \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} f(m,n) g^*(x+m, y+n) \tag{2.1}$$

In this equation, the images are represented as two-dimensional discrete functions (f and g). The $\circ$ operator is the cross correlation operator. The g^* function is the complex conjugate of g; this distinction is only important if the images are composed of complex numbers (as is actually the case with SAR imagery).

The output of the cross correlation is a data structure known as a correlation surface. Each point in the correlation surface represents a translation that could be applied to the second image. The translation is the position of the point relative to the center of the correlation surface. The surface's value at that point represents the level of correlation that would result if the second image were translated and compared to the first image.

Figure 2.3 is a typical correlation surface generated from two highly correlated images. Dark pixels correspond to low levels of correlation; light pixels correspond to high levels of correlation. The brightest pixel, which is slightly below and to the left of the center of the correlation surface, corresponds to the highest degree of correlation. For correlation surfaces generated from an image pair with a significant amount of common image content (for example, two different images of the same scene), the surface will resemble a delta function. It will have a single, distinctive, unambiguous peak. The surface will be smooth and continuous. The surface's values will sharply decrease as a function of their distance from the peak. Figure 2.4 provides a more intuitive visualization of the cross correlation, in the form of a three-dimensional surface plot.

This particular correlation surface is composed of 2048x2048 elements, the same dimensions as the input images. The peak value is 143 rows below the center and 2 columns to the left of the center. This means that if one were to translate the second input image by 143 rows down and 2 columns to the left, then the second input image would be perfectly centered with respect to the content of the first input image.

Finding the peak of the correlation surface yields the optimal translation that should be applied to the second image in order to get the best match with the first image. Of course, the correspondence between images is almost always going to be more complicated than a simple translation. The image registration algorithm uses information from cross correlations in a number of different ways, including the generation of affine transforms and (if desired) quadratic, cubic, or higher level transforms as well.

As noted earlier, the amount of processing time required to register an image pair is dominated by the efficiency of the algorithm used to generate cross correlations. The equation given above scales very poorly, as a direct implementation would be roughly $O(n^2)$. If the cross correlation is done in the frequency domain instead of the spatial domain, performance improves to $O(n + 3n \log n)$.

Cross correlation is essentially convolution, and one of the basic concepts in signal processing is that convolution in the spatial (or time) domain is equivalent to multiplication in the frequency domain. If $F(u, v)$ and $G(u, v)$ are the Fourier transforms or $f(x, y)$ and $g(x, y)$, then $f(x, y) \circ g(x, y)$ can also be derived by taking the inverse Fourier transform of $F(u, v)G^*(u, v)$. This is vastly more efficient (and scales much better) than the spatial domain equivalent given before [16, page 192].

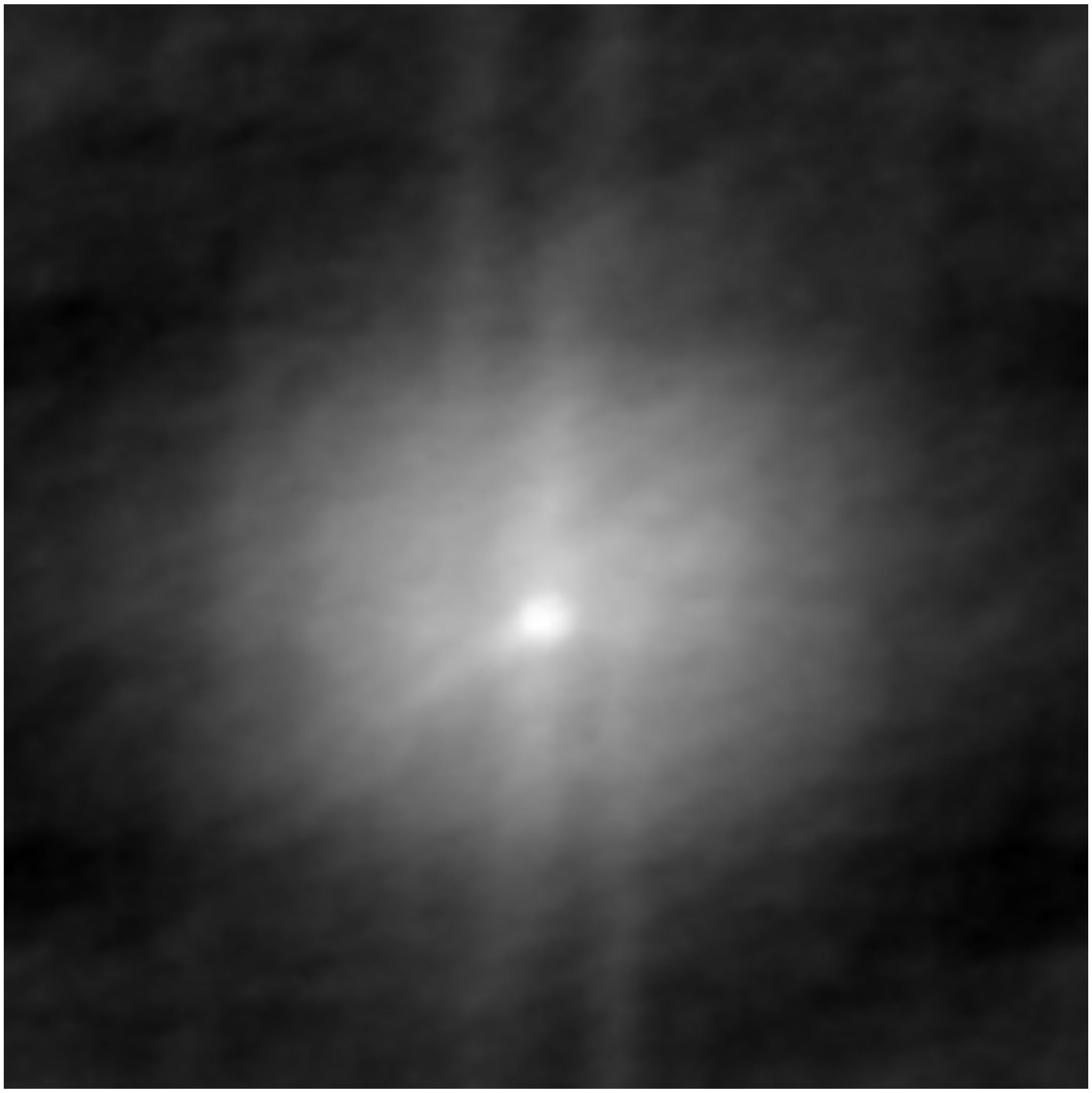


FIGURE 2.3: A correlation surface displayed as a 2D image.

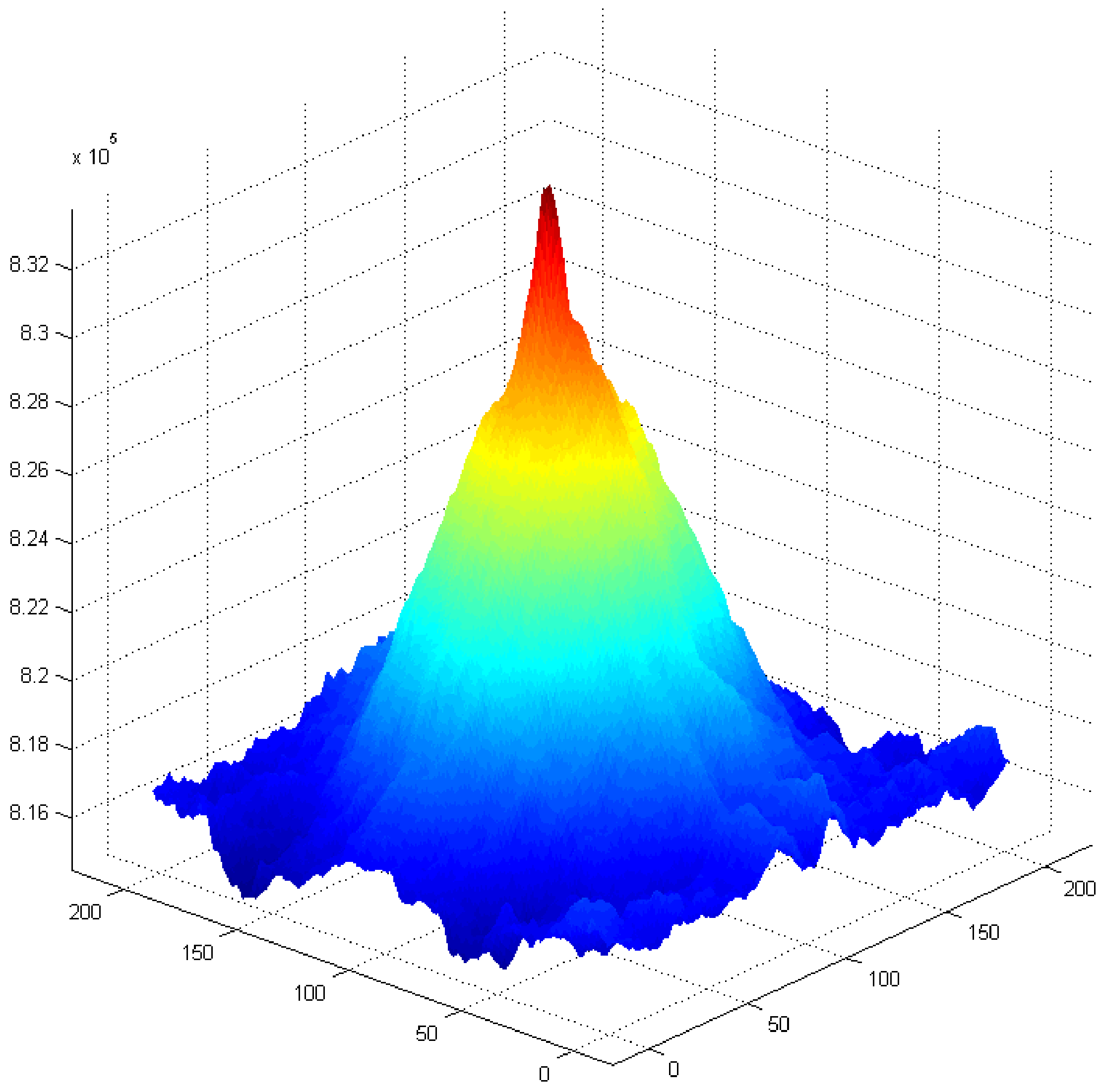


FIGURE 2.4: A correlation surface displayed as a 3D surface plot.

CHAPTER 3: THE IMAGE REGISTRATION PIPELINE

The following image registration algorithm is especially well-suited for airborne imaging applications. It assumes some *a priori* knowledge of the camera position and orientation, however it is flexible enough to allow for a small degree of error in this knowledge. Such errors are inevitable in airborne imaging.

The image registration algorithm presented below can be thought of as a pipeline — a series of operations in which each stage builds on the knowledge acquired in previous stages. The raw inputs to the pipeline are a pair of images, presumably depicting the same scene, and whatever data is required to roughly geocode both images. On modern imaging systems, this data will typically include the position and orientation of the aircraft at the time when each image was taken. This information is generated by the GPS/INS Kalman filter. The output of the image registration algorithm is a series of image transforms which, when applied one of the images (the warp image), will warp its pixel space into that of the other image (the base image). Ideally, after image registration, features in the base image and the transformed warp image will match up on a subpixel level.

3.1 Initial alignment

The first step in the image registration algorithm is the initial alignment phase. The inputs to this step are the raw input images, raw nav data (the output of the GPS/INS Kalman filter), and the geometric relationship between the camera and the aircraft. The output will be a version of the warp image which is roughly aligned with the base image, but not on a pixel level. Initial alignment registers the images using only what is known about the camera geometry; later steps in the pipeline will use image features to fine-tune the registration. The input images and the nav data are related via very precise timestamps. The nav data is typically generated on the order of 10 Hz, so some interpolation may be required.

3.1.1 Camera geometry

The nav data provides the position and orientation of the aircraft at a point in time. Of course, what is really needed is the camera's position and orientation, not the platform's. An additional transform will need to be applied which derives the camera's position and orientation from the aircraft's. If a gimbal is being used, this transform will not be constant; the orientation of the gimbal (and thus, the camera) relative to the aircraft can be obtained either from the gimbal's rotor decoder or by having an INS within the gimbal itself. Once this transform has been applied, we will have a description of a vector in world coordinates, with its origin at the camera, pointing along the optical axis of the camera towards the target.

3.1.2 The scene reference point

Now that we have a mathematical description of the camera's optical axis, we can use it to determine the *SRP* (scene reference point) of the image in question. The SRP of an image is the coordinate in 3D space at the image's center. This is needed in order to properly position the two images relative to each other, prior to registration.

Conceptually, the SRP is derived from the camera's position and orientation by intersecting the optical axis of the camera with the ground. If the range between the camera and the ground is known (typically using a laser rangefinder), then this derivation is trivial. Otherwise, it will be necessary to access a *DTED* (digital terrain elevation data) database in order to estimate the point of intersection. A DTED is simply a 2D array of elevations, sampled over a particular geographic region. The United States Department of Defense publishes a widely used standard for how DTED information is to be stored, and what levels of accuracy should be expected from them [18].

Even the act of intersecting the camera's optical axis with a DTED surface has a number of complications. To minimize error, it is generally a good idea to interpolate elevations between the samples present in the DTED database. Ideally this would be done using a spline surface generated from the DTED samples.

3.1.3 Vertical coordinate systems

Another complication in SRP generation is the fact that GPS systems report elevation differently from DTEDs. GPS receivers derive elevation in terms of *HAE* (height above ellipsoid), whereas DTEDs express elevation in terms of *MSL* (mean sea level). MSL represents elevation relative to the geoid, which is an equipotential surface of the Earth's gravitational field. The geoid is uneven, and known only by direct measurement and interpolation between measured points. The geoid roughly corresponds to sea level, hence the name. A large number of geoid models are in use today; there is no single standard geoid. The models differ from one another in the forms of measurement and interpolation used, as well as the region of the Earth covered by the model and the sampling distance.

HAE, on the other hand, represents elevation relative to a smooth, ellipsoidal approximation of the geoid. An ellipsoid is defined in terms of the lengths of its major and minor axes, and is concentric with the Earth's center of gravity. Measurements in HAE are more standardized than MSL, because while there is no single standard model of the geoid, there is a standard ellipsoid. This ellipsoid is referred to as the WGS84 ellipsoid, as it was defined as part of the World Geodetic System standard of 1984.

For purposes of SRP generation, either convert the DTED samples to HAE or convert the nav data to MSL. Since HAE is the more formally defined and consistent of the two systems, the author prefers to convert all vertical measurements to WGS84 HAE.

3.1.4 Horizontal coordinate systems

The final complication in SRP generation is the fact that neither the nav data nor the DTEDs are defined in terms of a Cartesian coordinate system. Internally, GPS receivers make all computations in terms of the *ECEF* (Earth-centered, Earth-fixed) coordinate system. ECEF is a 3D coordinate system, with units of meters. The origin of the coordinate system is the center of gravity of the Earth (hence, "Earth-centered"), and the axes of the coordinate system rotate with the Earth (hence, "Earth-fixed"). The positive x-axis is defined by a vector from the center of the Earth to the intersection of the equator and the Prime Meridian (0° latitude, 0° longitude). The positive z-axis is defined by a vector from the center of the Earth through the north pole, along the Earth's axis of rotation. The positive y-axis lies in the equatorial plane, and forms a right-handed coordinate system with the other two axes.

However, the output of the GPS receiver is typically in geodetic coordinates (latitude and longitude). ECEF coordinates are generally also available, but are unwieldy to work with as none of the planes of the ECEF system are aligned with the surface of the Earth (except for at the equator and the poles). DTED samples are also in terms of geodetic coordinates; the official DTED standard describes a DTED as a 2D array of values, in which the two axes represent latitude and longitude with a fixed spacing between each sample. The spacing is defined in terms of fractions of degrees, not linear distances.

In order to figuratively shoot a ray from the camera to the surface of the Earth, a 3D Cartesian coordinate system is required. Geodetic coordinates won't work because they are in a polar coordinate system. ECEF is theoretically acceptable, however in practice is difficult to work with due to its disassociation from the Earth's surface. A generally acceptable compromise is to use *UTM* (Universal Transverse Mercator) coordinates for the x and y axes, and HAE for the z axis.

UTM is a 2D Cartesian coordinate system which projects the surface of the ellipsoid onto a flat surface. The units are meters, and the two axes are referred to as *easting* and *northing*, because those are the directions they point in. UTM allows us to use linear measurements within the frame of reference of east and north. Of course, UTM is an approximation with inherent errors. These errors are minimized by customizing the projection in each of 60 UTM zones; each zone covers an area spanning 6° of longitude, and stretches from pole to pole. Furthermore, each zone is split between the northern and southern hemispheres. To fully specify a UTM coordinate, one needs to know the zone, the hemisphere (north or south), and the easting and northing. For all but the largest images, it is acceptable to use a single zone and hemisphere for the entire image, and distinguish between pixels using nothing but eastings and northings. The errors inherent in the UTM projection are generally well within the bounds acceptable to most airborne imaging applications, often on the order of centimeters or millimeters. The official specification for UTM is [17]. Snyder [15] is generally considered to be the ultimate resource for working with various geospatial coordinate systems (including UTM).

3.1.5 SRP generation

Once we have transformed the nav data into the camera's frame of reference, and transformed the camera geometry into a UTM/HAE-based 3D Cartesian coordinate system, we are ready to derive an SRP for each image. If we have a direct measurement of range from a laser rangefinder, then SRP generation should be trivial and highly accurate. Otherwise, it will be necessary to intersect the optical axis of the camera with the surface of the Earth, as represented in a database of DTEDs. The DTEDs should also be converted to the UTM/HAE system. For maximum accuracy, a continuous surface should be interpolated over the samples from the relevant DTED file, perhaps using 3D spline interpolation.

3.1.6 GSD

The GSD (ground sample distance) is the distance between adjacent pixels in physical units of length (typically meters). Knowing the two images' respective GSDs tells us the relative scale factor between the images. We can estimate the GSD of the image based on the distance between the camera and the SRP, as well as the camera's FOV (field of view) and image dimensions. Assuming uniform optics, determining the GSD is simple trigonometry:

$$GSD = \frac{r \tan FOV/2}{d} \tag{3.1}$$

Here, r is the range (the distance between the camera and the SRP), FOV is the field of view of the camera, and d is the breadth of the digitized image, in pixels. For cameras which produce square images, d is easy to determine (for a 2048x2048 image, d is 2048). For cameras with other aspect ratios, d is the axis which the field of view is defined in terms of. For example, a camera might be designed to generate HDTV-quality images, and produce 1920x1080 images. The optics are usually designed so that the generated images have "square pixels", *i.e.* the GSD will be the same along both image axes. By convention, a camera's specified FOV is generally defined in terms of the longer of the two axes; d will be 1920. Along the shorter axis, the FOV will be smaller but the GSD will be the same.

GSD might be further modified by cameras with large depression angles. In this context, the depression angle is the angle between the camera's optical axis and a vector normal to the ellipsoid at the camera's position. A camera with a depression angle of zero would be pointing directly at the center of the Earth, below the aircraft. A camera with a higher depression angle will image targets with a nonzero standoff range. The standoff range is the distance between the horizontal position (easting and northing only) of the aircraft and the horizontal position of the target, disregarding differences in elevation. The greater the depression angle, the greater the standoff range, and the greater the effects of perspective in the resultant image. For cameras at nadir (pointing straight down, having a near-zero depression angle), the images can be approximated as orthographic projections of the surface of the Earth. For higher depression angles, the effects of perspective and foreshortening come into play. Even with uniform optics, each image axis will have a different GSD. One axis will be consistent with the GSD equation, but the other axis will appear to be scaled as a function of the depression angle.

For the sake of simplicity of explanation, this thesis avoids dealing with the effects of off-nadir imaging, and focuses on the simpler case of nadir (or near-nadir) image acquisition. This will generally be the case in most civilian airborne imaging applications. Some of the main situations in which off-nadir imaging is required are military ISR (intelligence, surveillance, and reconnaissance) missions in which there are regions which are unsafe for the aircraft to fly directly over. In such situations, large standoff ranges are a safety requirement which override the need for optimal imaging geometries.

Another situation where higher depression angles are beneficial are stereo mapping missions, in which the topography of a region is determined by taking multiple images from varying perspectives. A higher depression angle exagerrates the geometric distortion caused by objects in a scene having different heights. If the goal is to generate a topographic map of an area (or simply to measure the height of specific objects, such as buildings or trees), then the distortion caused by off-nadir imaging is actually desirable, and a source of useful information about the scene.

3.1.7 Orientation

The SRP and GSD are two of the three pieces of information needed to perform initial alignment on a pair of images. The final piece of information is the orientation of each image. In this context, the orientation of an image is the angle of its negative y-axis (*i.e.*, "up" in the image) relative to north. This is simply the sum of the yaw of the aircraft (an angle relative to north) and the yaw of the camera relative to the aircraft.

3.1.8 Projection

The final stage of initial alignment is to project the warp image into the base image's pixel space based on each image's SRP, GSD, and orientation. Note that at this point, no action has yet been taken based on analysis of either image's content. The purpose of initial alignment is to make a preliminary best estimate as to how to register the images, which will set the stage for later fine-tuning of the registration using various image processing techniques.

At this point, we should have reasonable estimates of the data required in order to *geocode* both images. An image is *geolocated* if there is some known association between the scene in the image and a geographic position. For example, an image analyst might annotate an image by designating features in a scene with known positions. A specific point in an image with a known geographic position is referred to as a *tiepoint*; an image is geolocated if it has one or more tiepoints. Geocoding an image is a step beyond geolocation. Once an image has been geocoded, it is possible to determine the geographic position of every pixel, not just the pixels with associated tiepoints.

An image's geocoding is most commonly represented as an affine transform. The transform is a matrix which converts pixel coordinates in an image to some sort of projected coordinate system (typically UTM). In the case of UTM, the entire image would be associated with a single UTM zone and hemisphere, and the geocoding matrix would generate easting and northing values for each pixel. Applying a geocoding matrix to derive easting and northing from pixel coordinates is simply an affine transform. For example, to convert pixel (x, y) to the UTM coordinate (E, N), perform the following matrix multiplication:

$$\begin{bmatrix} \frac{dE}{dx} & \frac{dE}{dy} & T_E \\ \frac{dN}{dx} & \frac{dN}{dy} & T_N \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} E \\ N \\ 1 \end{bmatrix} \tag{3.2}$$

The geocoding matrix G is defined in terms of $\frac{dE}{dx}$ and $\frac{dE}{dy}$, the rates of change of easting with respect to the two image axes; $\frac{dN}{dx}$ and $\frac{dN}{dy}$, the rates of change of northing; and T_E and T_N, the easting and northing translation factors. Pixel coordinate $(0, 0)$ in the image has an easting and northing of (T_E, T_N).

Another useful matrix is G^{-1}, the inverse geocoding matrix. It can be used to determine the pixel coordinate of a given geographic coordinate. The inverse geocoding matrix is an invaluable tool in many geospatial image processing algorithms. After deriving the warp image's G^{-1} and the base image's G, we will be able to use an image processing operation known as a *geoblit* to project the warp image's pixel contents into the base image's geographic pixel space.

In the following equations, let the width and height of the image in pixels be denoted by W and H, respectively. For completeness, do not assume that the GSD is the same along both axes. Use GSD_x and GSD_y to refer to the GSD along the two axes. GSD shall always be positive. SRP_E and SRP_N refer to the easting and northing of the SRP. Let α refer to the angle of orientation of the image with respect to north. These equations assume that the origin of the pixel coordinate system is in the upper-left corner of the image, and that the x-axis increases to the right and the y-axis increases when moving downwards in the image. This is the standard coordinate system used in most image processing literature and software. When the negative y-axis points directly north, α is zero. Otherwise, α increases with clockwise rotation of the negative y-axis with respect to north. In other words, when α equals

0°, "up" in the image is north and "right" in the image is east. When α equals 90°, "up" is east and "right" is south. When α equals 180°, "up" is south and "right" is west. Finally, when α equals 270°, "up" is west and "right" is north.

$$\frac{dE}{dx} = GSD_x \cos\alpha \tag{3.3}$$

$$\frac{dE}{dy} = -GSD_y \sin\alpha \tag{3.4}$$

$$\frac{dN}{dx} = -GSD_x \cos\alpha \tag{3.5}$$

$$\frac{dN}{dy} = -GSD_y \sin\alpha \tag{3.6}$$

$$T_E = SRP_E - \frac{W}{2}\frac{dE}{dx} - \frac{H}{2}\frac{dE}{dy} \tag{3.7}$$

$$T_N = SRP_N - \frac{W}{2}\frac{dN}{dx} - \frac{H}{2}\frac{dN}{dy} \tag{3.8}$$

Use these equations to derive G for each image. Then, invert the warp image's G to obtain its G^{-1}. Now that we have the geocoding matrix G for the base image and G^{-1} for the warp image, we are ready to geoblit the contents of the warp image into the base image's geographic pixel space. This will complete the initial alignment stage of the registration pipeline.

A *bitblt* (bit block transfer), sometimes known as a *blit*, is a standard image processing operation in which some portion of an image is copied into another image. Typical the programmer identifies a source image, a rectangle within the source image, a destination image, and a point of origin. The pixels within the rectangle are cropped out of the source image and copied into the destination image. The point of origin is where the upper-left corner of the rectangle is mapped to in the destination image; essentially it applies a translation to the cropped rectangle while its pixels are being copied from the source to the destination.

A geoblit also copies pixels from a source to a destination image, but it uses the geocoding matrices of the two images to determine where to copy each pixel to in the destination. A pixel from the source image will be copied to the same geographic location within the destination image. It is always possible, of course, that that location does not lie within the bounds of the destination.

To complete initial alignment, we need to project the warp image into the base image's pixel space. First, create an image having the same dimensions and geocoding matrix as the base image. Initialize the new image's pixels to zero. Iterate through every pixel in the new image. For each pixel, use the new image's geocoding matrix (which is identical to that of the base image) to determine the UTM coordinates of that pixel. Use the warp image's inverse geocoding matrix to determine the location of the corresponding pixel in the warp image. Copy the pixel from the warp image to the new image, interpolating when necessary. After this process is complete, the new image will cover the exact same region of terrain as the base image, but it will contain the pixel data from the warp image. Pixels in the new image which were not present in the warp image will still have their initial value of zero (black).

The inputs to the initial alignment stage were the two original images and their associated nav data. The outputs (and thus, inputs to the next stage of registration) are the original base image, the projected warp image created by the geoblit operation, and the geocoding matrix that is now common to both images. All references to the warp image in the description of the next stage of registration (correlated alignment) refer to the projected warp image, not the original, unmodified warp image.

3.1.9 Initial alignment without nav data

Nearly all professional airborne imaging applications incorporate a Kalman-filtered GPS/INS system during image acquisition. However, it is sometimes necessary to register images for which there is no nav data. It will be clear to the user from visual inspection that a pair of images are different depictions of the same scene, but for whatever reason nav data is not available.

In this case, another approach to initial alignment is required. Rather than using the nav data to automatically scale, orient, and center the images, an interactive technique is required. The user can simply manually select three or more features which are common to both images, taking note of the pixel coordinates of the features in each image. The set pixel coordinates must be noncollinear for this technique to work.

The selected pixel coordinates of common features from both images can then be used to generate a least-squares solution for an optimal affine transform matrix. This is very similar to the technique used for correlated alignment; see the next section for detailed description of the actual mathematics involved. The initial alignment stage can be completed simply by applying this transform to the warp image.

It should be noted that if this technique is used, it is still almost always advantageous to go through the correlated alignment process, even if correlated alignment only involves creating another affine transform matrix. This is because there will no doubt be some user error in selecting the correct pixel for a given feature in both images. Furthermore, if the scene contains a lot of relief (fluctuations in elevation), then there is a high probability that using a small number of features for registration (3–5 is typical for interactive applications) will result in a suboptimal transform matrix. Performing correlated alignment after manual initial alignment will help iron out errors in the user's pixel selections, as well as average out errors due to image distortions caused by the scene's topography.

3.2 Correlated alignment

The initial alignment stage is a first approximation of the transforms needed to register the warp image to the base image. A fundamental difference between initial alignment and correlated alignment is, initial alignment uses what is known about the camera geometry and the terrain topography to register the images. It does not employ any sort of pattern-matching using the image pixel data. Correlated alignment refines this initial approximation via analysis of the pixel data itself.

The inputs to correlated alignment are the outputs from initial alignment: the untouched base image, and the modified warp image. The warp image should be centered, scaled, rotated and cropped to a reasonable approximation of the base image's geometry. The goal of correlated alignment is to compute, using image processing techniques, an optimal global transform that will best align the modified warp image with the base image. Essentially, this is a fine-tuning of the approximate registration produced in the initial alignment phase.

After correlated alignment, the differences between the base and warp images due to global parameters (such as camera geometry) should be minimized. Correlated alignment irons out the image alignment errors due to inaccurate nav data, inaccurate range estimates, vibration of imaging platform, errors in the timestamps given to images, and a multitude of other error sources inherent in modern airborne imaging systems.

The correlated alignment algorithm is as follows. A pair of images, I and I', contain slightly different views of the same scene. Many features that are visible in I are also visible in I'. Say that some number n points of correspondence have been identified between the two images. "Points of correspondence" are here defined as a pair of image coordinates, (x, y) from I and (x', y') from I', which designate the locations of some object or feature that is visible in both images.

First, assume that there is some ideal linear transform that maps all such points from I to their corresponding points in I':

$$\begin{bmatrix} a_x & b_x & c_x \\ b_y & a_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} \tag{3.9}$$

In the transform matrix, a_x and a_y are the x and y scale factors, b_x and b_y are the x and y shear factors, and c_x and c_y are the x and y translation factors. The matrix multiplication can be restated in the form of a pair of linear equations:

$$\begin{aligned} a_x x + b_x y + c_x &= x' \\ a_y y + b_y y + c_y &= y' \end{aligned} \tag{3.10}$$

To determine an optimal values for a_x, b_x, and c_x, first list the algebraic expressions for all known x':

$$\begin{aligned} a_x x_0 + b_x y_0 + c_x &= x'_0 \\ a_x x_1 + b_x y_1 + c_x &= x'_1 \\ a_x x_2 + b_x y_2 + c_x &= x'_2 \\ &\vdots \\ a_x x_n + b_x y_n + c_x &= x'_n \end{aligned} \tag{3.11}$$

Observe that this is equivalent to the following matrix expression:

$$\begin{bmatrix} x_0 & y_0 & 1 \\ x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ \vdots & & \\ x_n & y_n & 1 \end{bmatrix} \begin{bmatrix} a_x \\ b_x \\ c_x \end{bmatrix} = \begin{bmatrix} x'_0 \\ x'_1 \\ x'_2 \\ \vdots \\ x'_n \end{bmatrix} \tag{3.12}$$

The expression is conveniently in the form of $Ax = b$, in which the unknowns are collected within the x matrix. Using the pseudoinverse to find a least-squares solution to the system and evaluating $A^{-1}b$ will yield the optimal values for the x matrix, which contains a_x, b_x, and c_x. Repeating the process for the y coordinates will likewise yield optimal values for a_y, b_y, and c_y.

The remaining question is, how are the points of correspondence obtained in the first place? One possibility is to have a human image analyst identify them explicitly, by selecting the corresponding points from a pair of displayed images. There must be at least three points (or four, if a least-squares solution is desired), but there is no maximum number of points. The more points are selected, the greater the accuracy of the least-squares solution.

If the images do not have any geolocation information associated with them, then initial alignment (the previous stage in the registration algorithm) will be performed using this very technique. The user selects three or more noncollinear points of correspondence in the two images. Then, equation 3.12 is used to generate an optimal affine transform which maps the selected pixel coordinates in the warp image to the corresponding pixel coordinates in the base image. This transform matrix is then applied to the warp image. This is a suboptimal technique because it requires human intervention; it is much more efficient to perform initial alignment automatically using nav data to geolocate the images and generate the initial transform for the warp image. However, in situations where nav data is unavailable, it is the only alternative.

In correlated alignment, an automated system is used to locate points of correspondence. The same algorithm is used for correlated alignment regardless of which algorithm was used for initial alignment. Assuming that the two images are somewhat closely aligned already, subdivide the warp image into a number of smaller tiles. For example, chop up a 2048x2048 image into a 16x16 array of 128x128 pixel tiles. For each tile, extract the corresponding tile from the base image, and generate a correlation surface for the pair of tiles. Finding the peak of the correlation surface will yield an optimal translation between the two tiles which will best match them up. Determine the pixel coordinates of the center of each tile in terms of the warp image's pixel space: in the example given, the upper-left tile's center pixel would be $(64, 64)$; the tile to the right would be $(192, 64)$; the tile below would be $(64, 192)$, and so on. These coordinates are collectively (x_0, y_0) through (x_n, y_n) in equation 3.12. Adding the translations from the correlation surfaces, $(\Delta x_0, \Delta y_0)$ through $(\Delta x_n, \Delta y_n)$, yields the corresponding points in the base image, (x'_0, y'_0) through (x'_n, y'_n).

The only problem with the automated method is that care must be taken to filter out correlation surfaces which do not contain useful information. There are a number of image, sensor, or scene-dependent properties which might cause some of the tiles to not sufficiently correlate, or to have high degrees of correlation in unexpected areas. If these correlations are not filtered out, then any translations derived from the resultant surfaces will simply add noise and error to the final least squares solution. There are a number of indicators of poor correlations. Some commonly used indicators include:

- Correlation surfaces in which the peak value occurs on the leftmost or rightmost column, or the top or bottom row, are almost certainly invalid. In some cases, it may be worthwhile to discard correlations in which the peak is outside the central quadrant of the surface. The downside to this is the fact that restricting the valid area of the correlation surface also limits the maximum translation that can be detected.

- Count the number of black (zero-valued) pixels in the tile from the warp image that was used as an input to the correlation surface. Black pixels occur in areas where the original warp image (prior to initial alignment) and the base image don't have any overlap. The higher the percentage of black pixels in the tile, the less real information was used to generate the correlation surface. Discard any correlation surface in which this black pixel percentage is above some threshold.

- Subtract the minimum value in the correlation surface from every element in the correlation surface. This will remove any DC bias present due to the input images' spectrum. Then, compute the ratio of the peak value to the RMS (root mean square) of the entire surface. If the ratio is below some predetermined threshold, assume the correlation surface is noise and discard it.

Many other indicators can be used as well. Unfortunately, the specific set of indicators used to filter out bad correlations will be specific to the types of sensors used and the types of scenes being imaged. An effective set of indicators for a particular SAR might not work as well for an infrared imaging system. Some experimentation will likely be necessary to determine an ideal set of indicators for a particular application. In the implementation described in the next chapter, the only indicators used were detecting peaks in the first or last row or column, and rejecting any correlation surface in which the input tile from the warp image had more than 5% black pixels. In the author's experience, examining the peak:RMS ratio is absolutely essential when working with SAR images; however, for the optical dataset used in this work, it was just not useful.

It should be noted that the correlated alignment algorithm is not limited to generating affine transforms. It can just as easily be used to generate quadratic or cubic transforms, or any order desired. For example, in the quadratic case, the equations which map x to x' would look like this:

$$\begin{gathered}
a_x x_0^2 + b_x x_0 + c_x x_0 y_0 + d_x y_0 + e_x y_0^2 + f_x = x'_0 \\
a_x x_1^2 + b_x x_1 + c_x x_1 y_1 + d_x y_1 + e_x y_1^2 + f_x = x'_1 \\
a_x x_2^2 + b_x x_2 + c_x x_2 y_2 + d_x y_2 + e_x y_2^2 + f_x = x'_2 \\
\vdots \\
a_x x_n^2 + b_x x_n + c_x x_n y_n + d_x y_n + e_x y_n^2 + f_x = x'_n
\end{gathered} \tag{3.13}$$

The corresponding matrix equation would be constructed like so:

$$\begin{bmatrix} x_0^2 & x_0 & x_0y_0 & y_0 & y_0^2 & 1 \\ x_1^2 & x_1 & x_1y_1 & y_1 & y_1^2 & 1 \\ x_2^2 & x_2 & x_2y_2 & y_2 & y_2^2 & 1 \\ \vdots & & & & & \\ x_0^2 & x_n & x_ny_n & y_n & y_n^2 & 1 \end{bmatrix} \begin{bmatrix} a_x \\ b_x \\ c_x \\ d_x \\ e_x \\ f_x \end{bmatrix} = \begin{bmatrix} x_0' \\ x_1' \\ x_2' \\ \vdots \\ x_n' \end{bmatrix} \tag{3.14}$$

There are more coefficients to solve for, however the basic algorithm is identical to the affine case. In general, higher-order transforms are rarely necessary for airborne images. They are much more common in the case of satellite images, due to the curvature of the Earth, as well as the nature of the sensors themselves. Many satellite imaging systems in use today are pushbroom scanners, in which the sensor is a 1D array of detector elements. The sensor images one row of pixels at a time, which must be integrated into a complete image in postprocessing. This may necessitate the usage of higher order transforms, in order to remove geometric artifacts of the pushbroom imaging process.

CHAPTER 4: ANALYSIS

The algorithms described in chapter 3 were implemented and tested. The test data was primarily narrow field of view (NFOV) optical images of desert scenes, however some urban scenes were used as well. In addition, several wide field of view (WFOV) image pairs were tested, as well as several color images. In all cases the software was extremely effective in registering the images to a subpixel level. The software itself is described in appendix 4.4, and the source code is included in appendix 4.4.

4.1 Algorithms used

There was no nav data available for any of the images, therefore the alternative method of initial alignment was used. Three noncollinear control points were selected in the base image for each pair, and the pixel coordinates were found for the corresponding points in the warp image. This resulted in very good initial alignment for all image pairs. However, the initial alignment was by no means anywhere near optimal, as the correlated alignment stage always generated transforms which vastly improved the quality of the registration.

For correlated alignment, the images were divided up into 256x256 pixel tiles, however the tiles overlapped with their neighbors by 50%. So, starting in the upper left corner of the images, the first tiles were centered at pixel coordinate $(128, 128)$, and contained pixels from rows 0 through 255, and columns 0 through 255. The next pair of tiles were centered at pixel coordinate $(256, 256)$, and contained pixels from rows 0 through 255, and columns 128 through 513. Having 50% overlap between adjacent tiles allowed a larger correlation window to be used (which provides a larger search area and a higher resistance to noise), while providing up to four times as many points of correspondence as there would have been had the tiles not overlapped.

4.2 Evaluation techniques

4.2.1 Difference images

Several techniques were used to evaluate the output of the registration algorithm. Difference images were created after initial alignment and after correlated alignment in order to quantify the quality of the registration, and to determine the degree to which correlated alignment improved upon initial alignment. A difference image is generated simply by subtracting one image's pixel values from the corresponding pixels in the other image. All images were 8-bit grayscale, so the minimum pixel value was 0 and the maximum was 255. This means that the minimum possible value for a pixel in the difference image is -255 (minimum minus maximum), and the maximum possible value was 255 (maximum minus minimum).

Prior to image subtraction, each image's pixels were converted to floating point and normalized; all pixels were divided by 255 to change the range of possible pixel values to 0.0 to 1.0. After subtraction, the difference image's range was -1.0 to 1.0. The equation

$$y = (\frac{x}{2} + 0.5) * 255 \tag{4.1}$$

was then used to convert the difference image's pixels back to a range of 0 to 255. If a pixel's value was the same in the warp image as it was in the base image, then the corresponding pixel in the difference image would have a value of 127. If a pixel's value was different between the base and warp images, then the corresponding pixel in the difference image would be some amount above or below 127. If the images were perfectly registered, and there were no changes in the scene between the two images, then the ideal difference image would be perfectly gray and featureless. Every pixel would be 127. Of course, this never occurs in practice, but it's possible to measure how close a given difference image is to this ideal.

Figure 4.1 shows a pair of images prior to registration. Note that the aircraft was apparently flying in opposite directions when the two images were taken. Figure 4.2 shows the final registered image pair. Once again, the base image is on top and is unaltered by the process. The warp image is on the bottom, with black pixels assigned to nonoverlapping regions.

Figure 4.3 is the difference image that was generated using the output of the initial alignment stage. Recall that regions of the difference image which are grey, flat, and featureless imply good registration. If the registration is off by only a few pixels, then the difference image takes on the appearance of an out-of-focus, ghostly reflection of the original scene. Around the control points used to generate the transform, the registration is of very high quality. Further away from the control points, the registration rapidly deteriorates.

Figure 4.4 is the final difference image, generated after the correlated alignment stage. It shows a significant improvement over the previous difference image. The difference image has lost its out-of-focus appearance, and appears to have snapped into alignment across the entire image. At this point, any elements of the difference image which deviate from the uniform gray-value of 127 are due to changes in the scene rather than errors in registration. Imaging at different times of day can produce such deviations, as the shadows are cast in different directions in the two images. Variations in optical focus or aircraft altitude can also introduce minor changes in the difference image as well. However, these types of artifacts tend to produce very subtle deviations in the difference image which are globally visible throughout the image.

In general, the most visible (as well as the most localized) fluctuations remaining in the difference image after correlated alignment are due to one of two things, each of which leads to a useful application of image registration. The first cause is actual change in the scene, either from human activity or natural events. Algorithms which try and isolate these fluctuations in difference images are known as change detection algorithms. The second cause is from changing elevation in the scene. The effects of varying elevation in a scene are amplified when the two images are taken from different positions; the change in perspective causes the scene to look different in the two images, which results in deviations in the difference image. The registered images can then be passed on to algorithms which further refine the registration, and in the process measure the topography of the terrain [9].

Both of these applications have initial and correlated alignment as prerequisites, but their algorithms diverge at that point and go off in very different directions. Mission planning figures prominently in both cases. In the case of change detection, the imaging geometry is set up in such a way that the imaging platform has a position and orientation on the second flight that is nearly identical to the that of first flight. This minimizes changes due to terrain geometry and ensures that the most notable fluctuations in the difference image are indeed due to changes in the scene. Mapping missions, on the other hand, are set up so that each image attempts to capture the scene from as widely varying imaging geometries as possible. This is intended to emphasize the geometric distortion in the image pairs due to changing elevation.

4.2.2 Control point migration

Difference images provide an excellent visual representation of the quality of the registration (as well as a useful input to other algorithms), however it can be difficult to quantify the quality metric that they represent. For more formal analysis, monitoring control point migration can be very useful technique. Control point migration is simply the movement of the control points (the points of correspondence identified during correlated alignment) during the final affine transform.

Each control point has three sets of pixel coordinates associated with it. The first coordinate is the center of the control point's correlation window in the base image. For example, in the scheme described in section 4.1, the images were chopped up

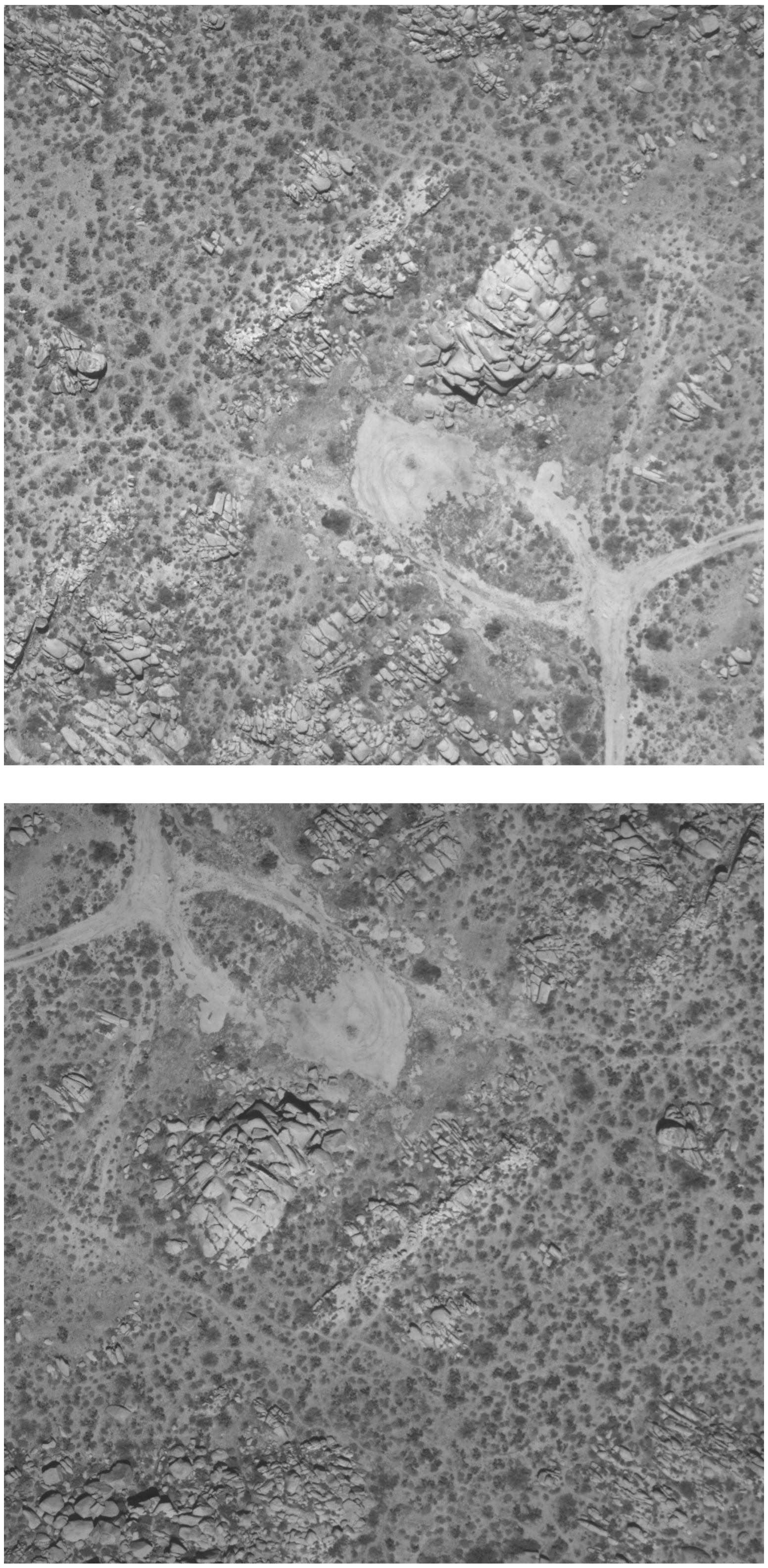


FIGURE 4.1: Another image pair prior to registration.

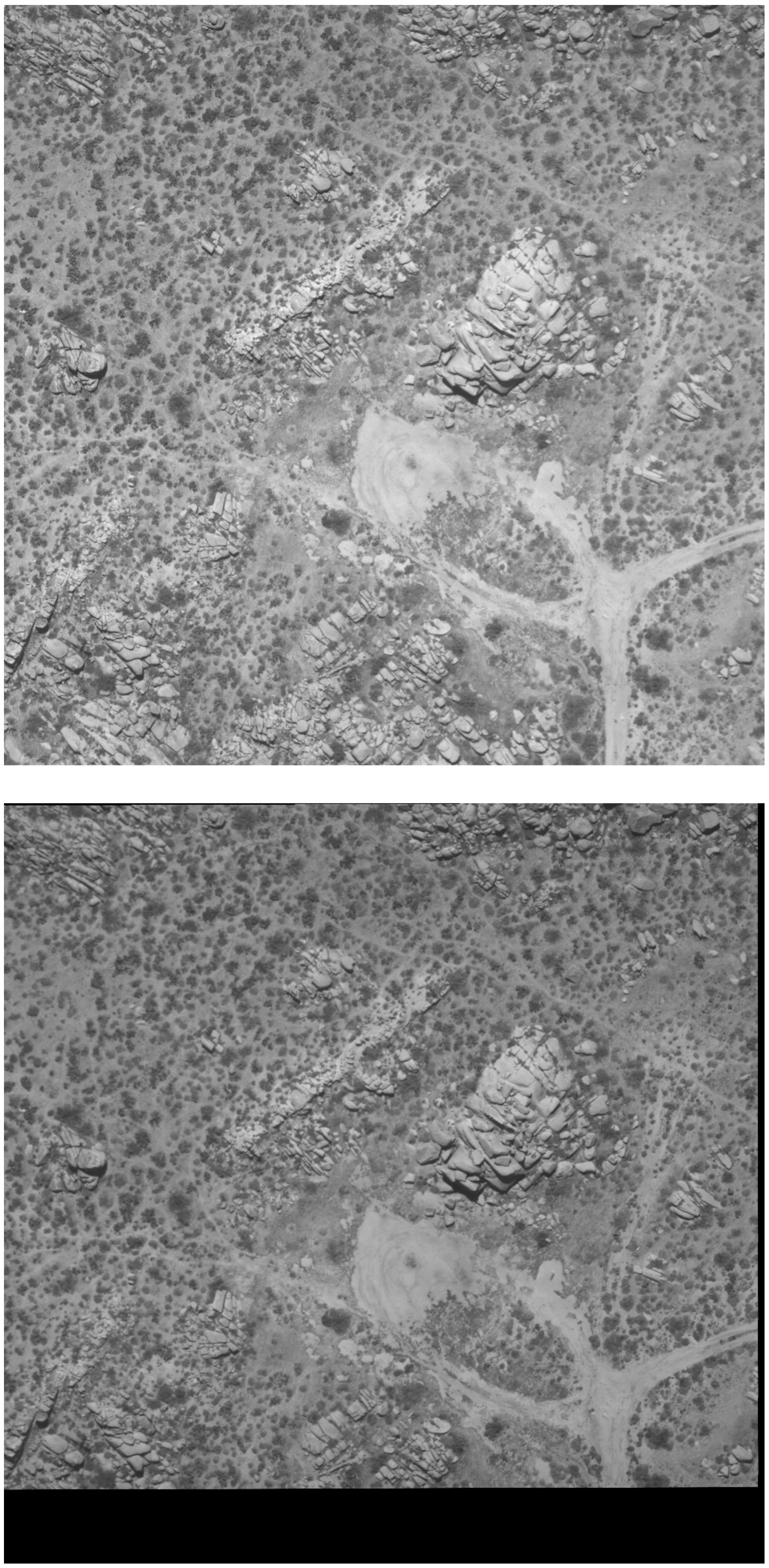


FIGURE 4.2: The image pair after registration.

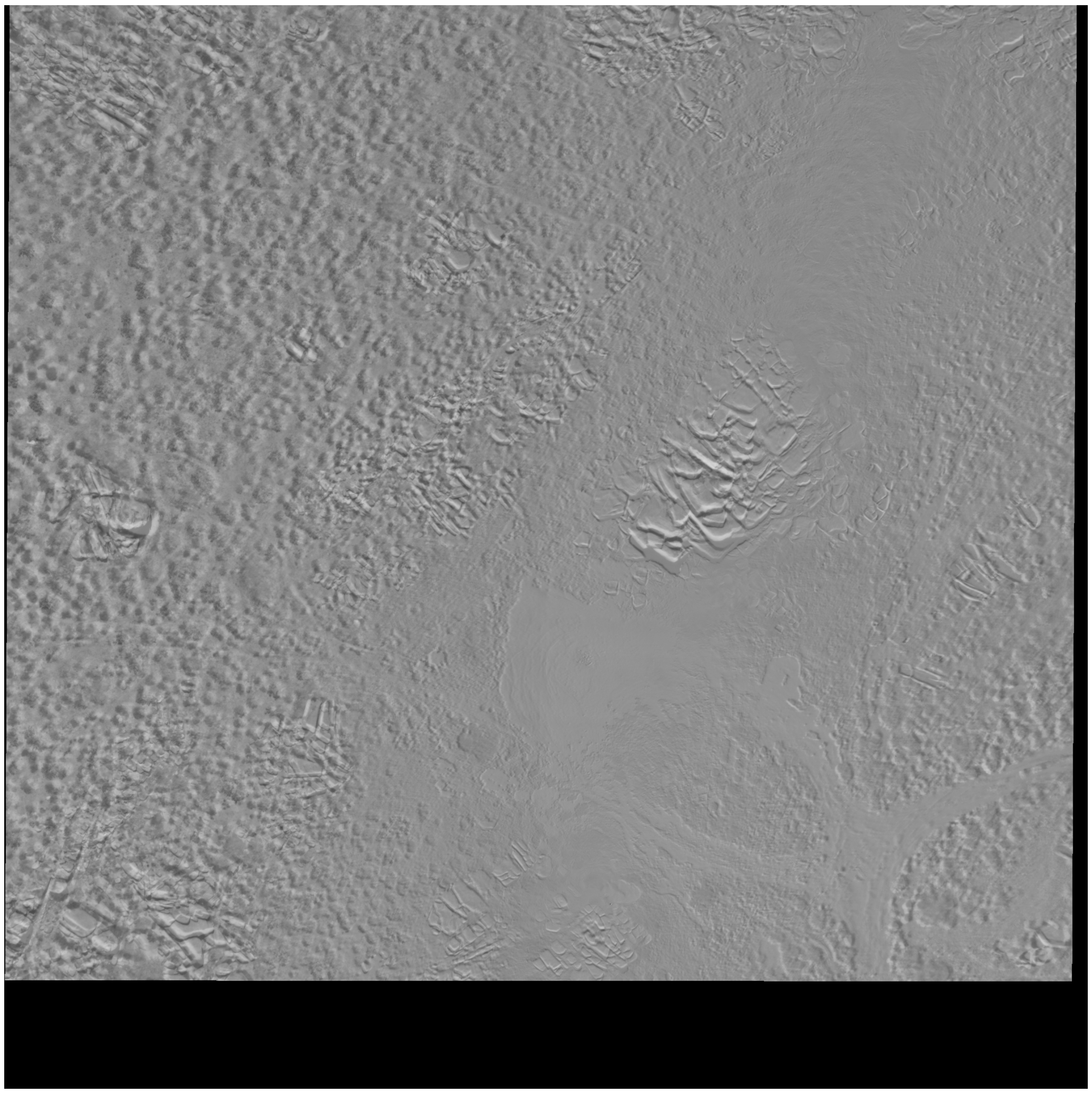


FIGURE 4.3: The difference image after initial alignment.

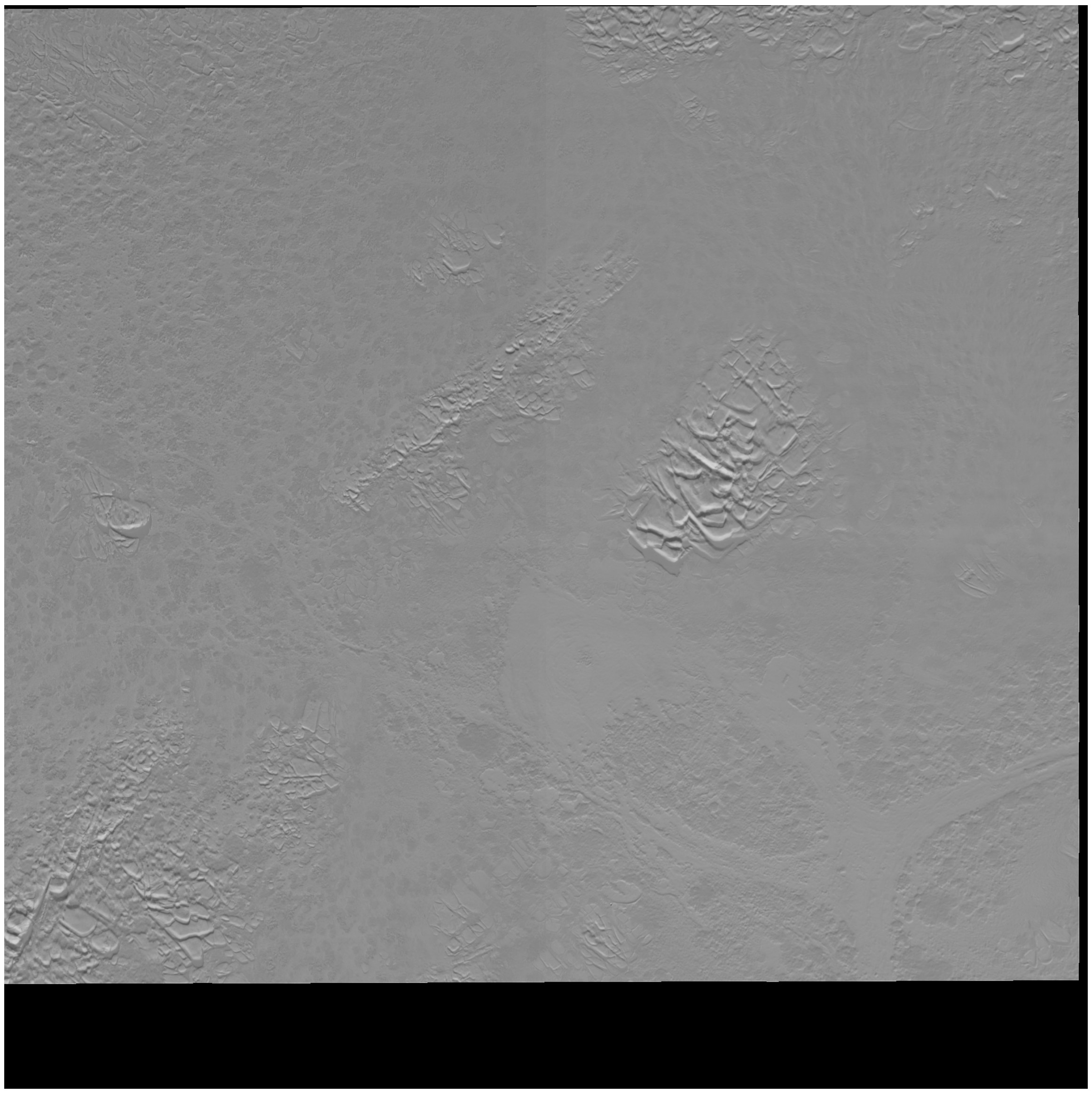


FIGURE 4.4: The difference image after correlated alignment.

into 256x256 pixel tiles which overlapped by 50% with neighboring tiles. The centers of these tiles (or correlation windows) occurred at $(128, 128)$, $(256, 128)$, $(384, 128)$, and so on in the base image. These centers are the first coordinate associated with a control point.

The second coordinate is the corresponding point found in the warp image. Recall that each tile in the base image was cross correlated with the corresponding tile in the warp image. The result of the cross correlation is a translation that would center the warp tile with respect to the base tile. Another way of thinking about this is, the result of the cross correlation is the identification of a specific pixel in the warp tile that maps to the center pixel of the base tile. The coordinate of this pixel in the warp image is the second coordinate associated with a control point.

After correlated alignment, an optimal affine transform is computed and applied to the warp image. By applying this transform to coordinates just described (the control points in the warp image), we can see how closely each control point in the warp image gets mapped to its corresponding control point in the base image. We can measure the Cartesian distance between each control point in the warp image to its matching point in the base image, then measure that distance again after applying the final affine transform. If the registration algorithm is working correctly, that distance should noticeably decrease after the transform.

The distances are also useful as predictors for the effectiveness of change detection algorithms or terrain modeling algorithms. These algorithms generally require sub-pixel registration in order to work properly. If the final transform fails to bring most of the warp control points to within a pixel of their corresponding base control points, then the algorithms which operate on the registered images will be less reliable.

Table 4.1 contains a selection of the control point migration statistics for the image pair shown in this chapter. The first three columns correspond to the set of three pixel coordinates just described: “base” is the center pixel of a given base image tile, “warp”

base	warp	warped	dist0	dist1
$(128, 128)$	$(150, 123)$	$(129, 128)$	22.6	1.0
$(256, 128)$	$(276, 123)$	$(257, 128)$	20.6	1.0
$(384, 128)$	$(402, 125)$	$(384, 129)$	18.2	1.0
$(512, 128)$	$(528, 125)$	$(512, 129)$	16.3	1.0
$(640, 128)$	$(654, 126)$	$(640, 129)$	14.1	1.0
$(768, 128)$	$(781, 126)$	$(768, 129)$	13.2	1.0
$(896, 128)$	$(907, 126)$	$(896, 128)$	11.2	0.0
$(1024, 128)$	$(1033, 126)$	$(1023, 127)$	9.2	1.4
$(1152, 128)$	$(1159, 125)$	$(1151, 126)$	7.6	2.2
$(1280, 128)$	$(1286, 125)$	$(1280, 125)$	6.7	3.0
$(1408, 128)$	$(1412, 126)$	$(1407, 126)$	4.5	2.2
$(1536, 128)$	$(1538, 127)$	$(1535, 126)$	2.2	2.2
$(1664, 128)$	$(1664, 128)$	$(1662, 127)$	0.0	2.2
$(1792, 128)$	$(1791, 127)$	$(1791, 125)$	1.4	3.2
$(128, 256)$	$(149, 252)$	$(129, 257)$	21.4	1.4
$(256, 256)$	$(275, 252)$	$(256, 257)$	19.4	1.0
$(384, 256)$	$(401, 253)$	$(384, 257)$	17.3	1.0
$(512, 256)$	$(527, 253)$	$(512, 257)$	15.3	1.0
$(512, 256)$	$(527, 253)$	$(512, 257)$	15.3	1.0

TABLE 4.1: A portion of the control point migration table.

is the corresponding pixel in the warp image, and “warped” is the warp coordinate’s position after the final transform has been applied. The next two columns are the distance between “base” and “warp”, and the distance between “base” and “warped”, respectively. It should be noted that although “warped” coordinates are listed as having integral values for the sake of brevity, in reality they are real-valued. The image registration algorithm described here operates at a sub-pixel level of precision. It nearly always maps pixels to fractional coordinates, necessitating interpolation when the transforms (in all stages) are applied to the warp image.

4.3 Additional imagery

This section contains additional examples of image registration. Three image pairs are presented. For each image pair, the following figures and tables are provided:

base	warp	warped	dist0	dist1
$(256, 256)$	$(257, 257)$	$(255, 254)$	1.4	2.2
$(384, 256)$	$(385, 256)$	$(384, 254)$	1.0	2.0
$(512, 256)$	$(512, 256)$	$(511, 256)$	0.0	1.0
$(640, 256)$	$(640, 254)$	$(640, 255)$	2.0	1.0
$(768, 256)$	$(767, 253)$	$(768, 255)$	3.2	1.0
$(896, 256)$	$(894, 253)$	$(895, 257)$	3.6	1.4
$(256, 384)$	$(257, 386)$	$(255, 383)$	2.2	1.4
$(384, 384)$	$(385, 385)$	$(384, 383)$	1.4	1.0
$(512, 384)$	$(512, 384)$	$(511, 383)$	0.0	1.4
$(640, 384)$	$(640, 383)$	$(640, 384)$	1.0	0.0
$(768, 384)$	$(767, 382)$	$(767, 384)$	2.2	1.0
$(896, 384)$	$(894, 381)$	$(895, 384)$	3.6	1.0
$(256, 512)$	$(257, 515)$	$(255, 512)$	3.2	1.0
$(384, 512)$	$(385, 514)$	$(383, 512)$	2.2	1.0
$(512, 512)$	$(513, 512)$	$(512, 511)$	1.0	1.0
$(640, 512)$	$(640, 511)$	$(639, 512)$	1.0	1.0
$(768, 512)$	$(768, 510)$	$(768, 512)$	2.0	0.0
$(896, 512)$	$(895, 509)$	$(896, 512)$	3.2	0.0
$(896, 512)$	$(895, 509)$	$(896, 512)$	3.2	0.0

TABLE 4.2: A portion of the control point migration table from pair 1.

- A comparison of the unmodified base and warp images.
- A comparison of the base image and the registered warp image.
- A difference image generated after the initial alignment stage.
- A difference image generated after the correlated alignment stage.
- A selection from the control point migration table generated while registering the image pair.

The control point migration table is not printed in its entirety, as it would require 15–20 pages to print the complete table for a single image pair. The samples in the table are representative of the entire dataset, however.

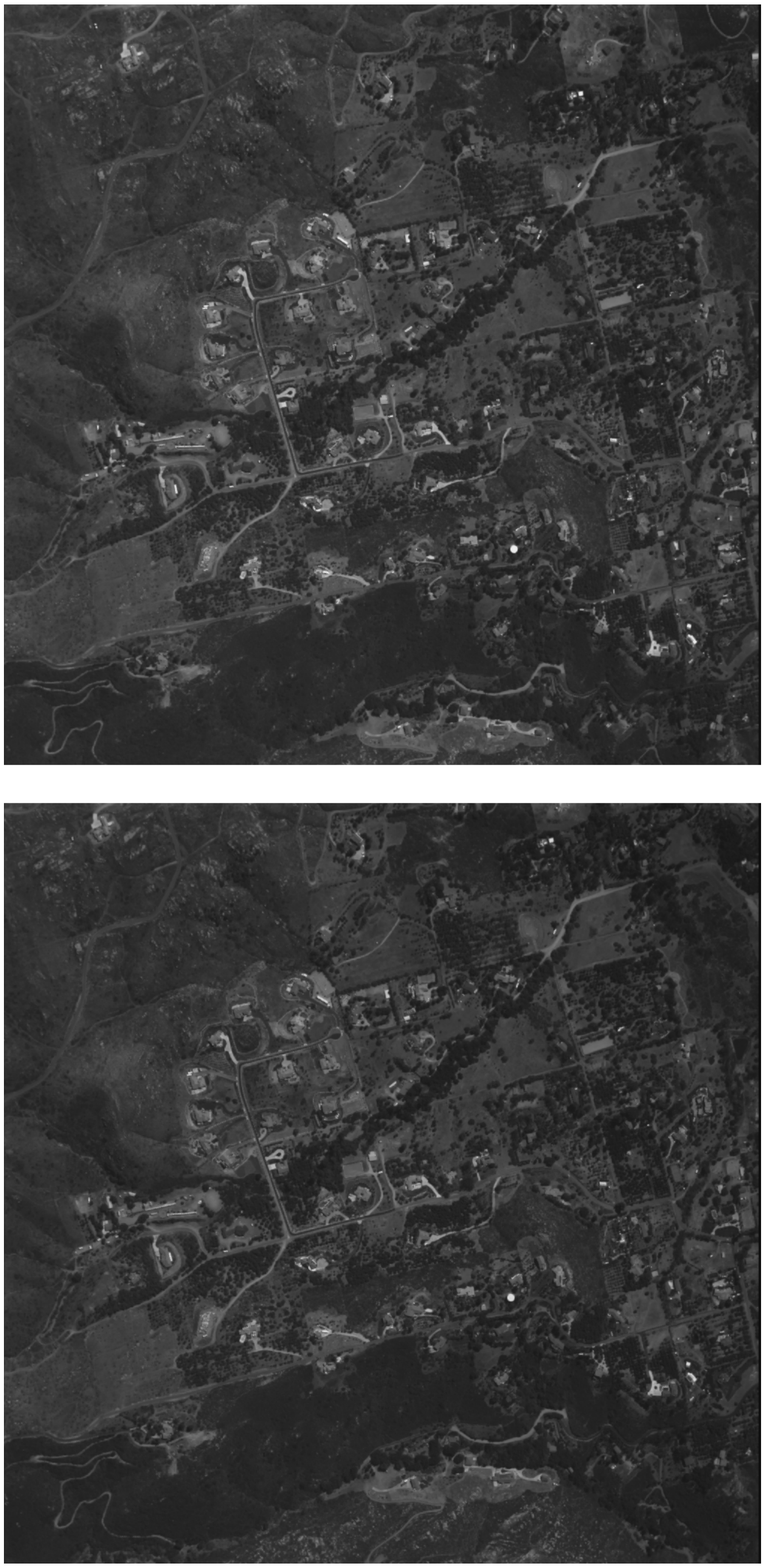


FIGURE 4.5: Image pair 1, prior to registration.

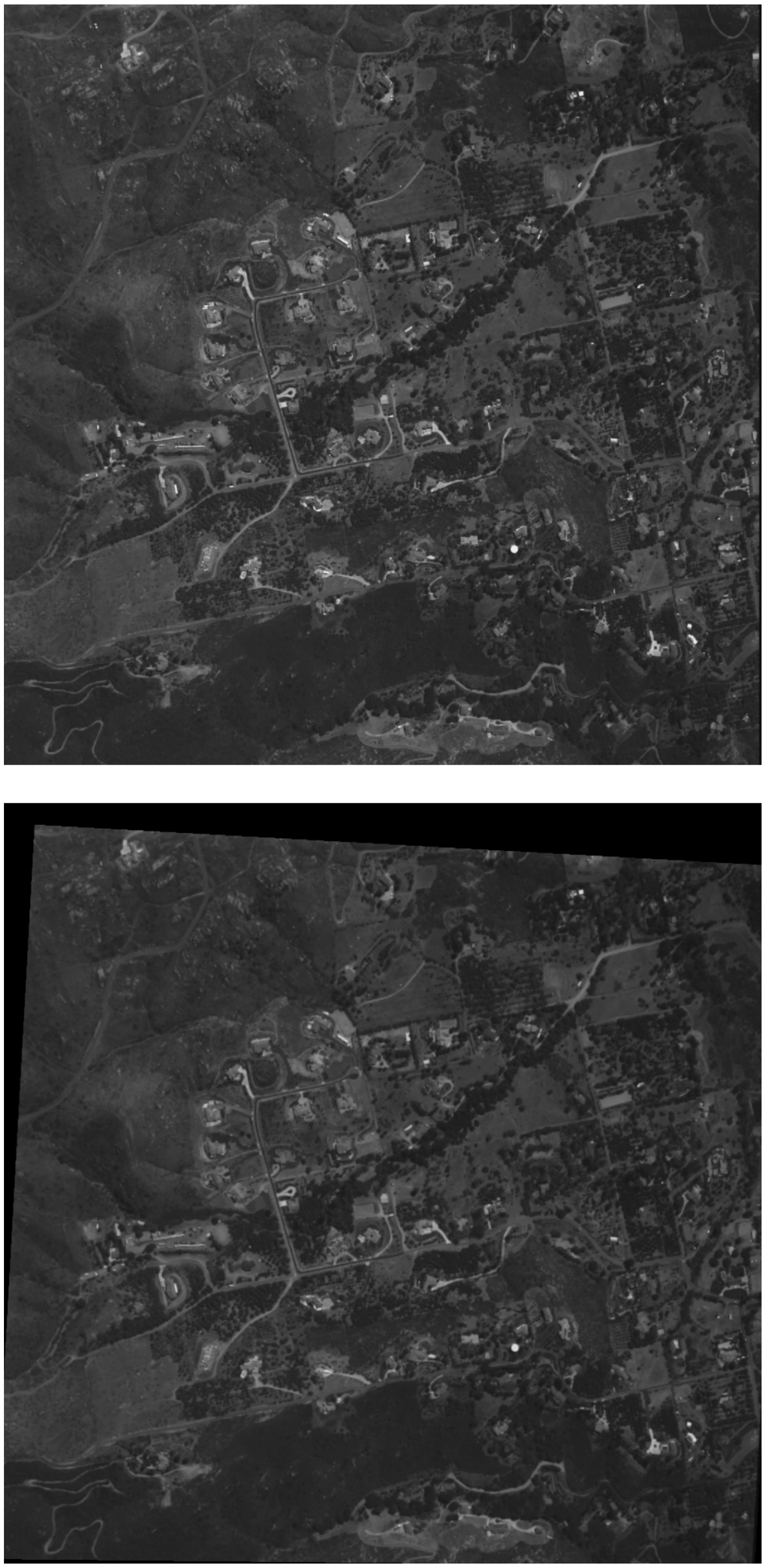


FIGURE 4.6: Image pair 1, after registration.

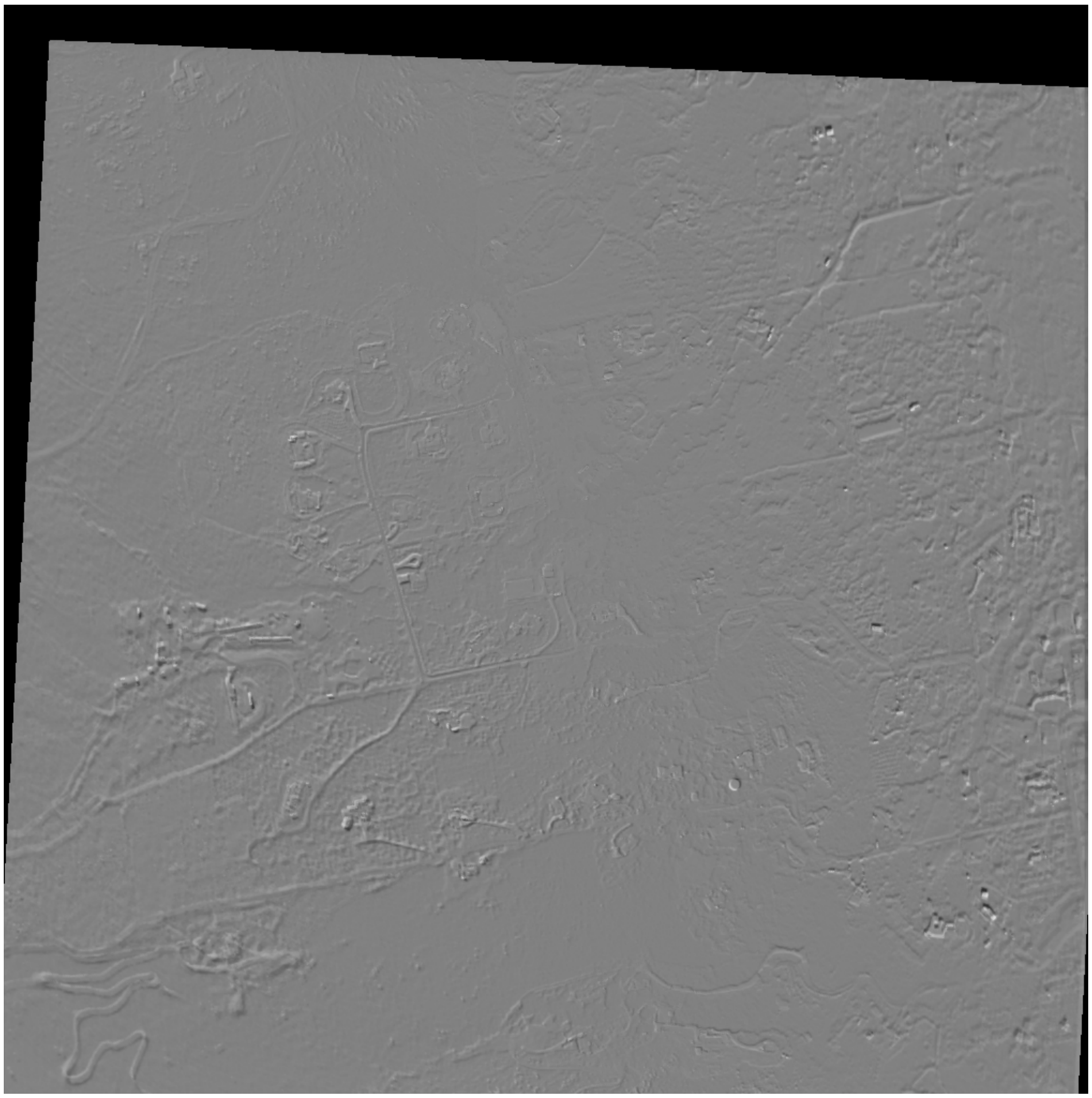


FIGURE 4.7: The difference image from pair 1, after initial alignment.

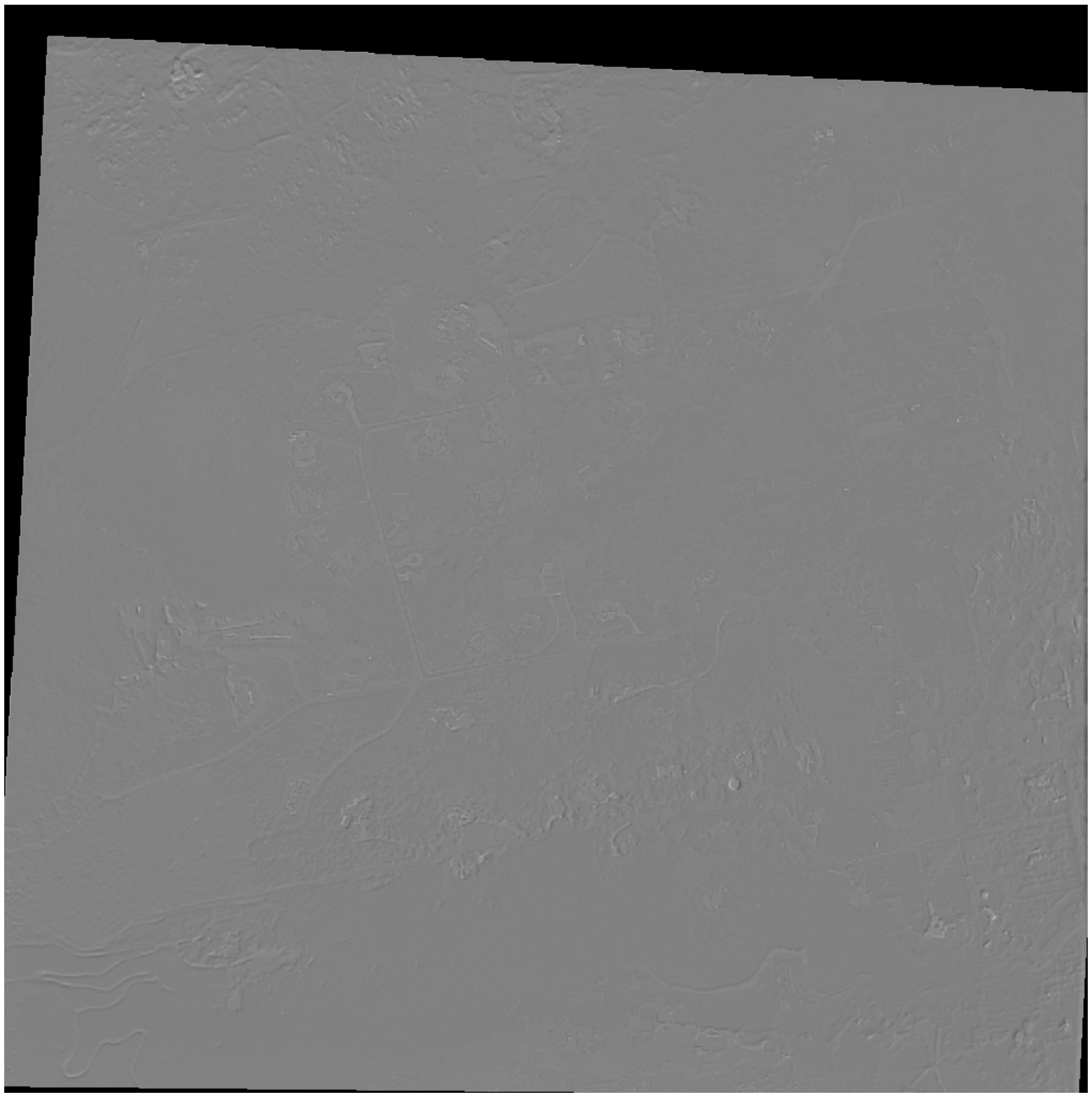


FIGURE 4.8: The difference image from pair 1, after correlated alignment.

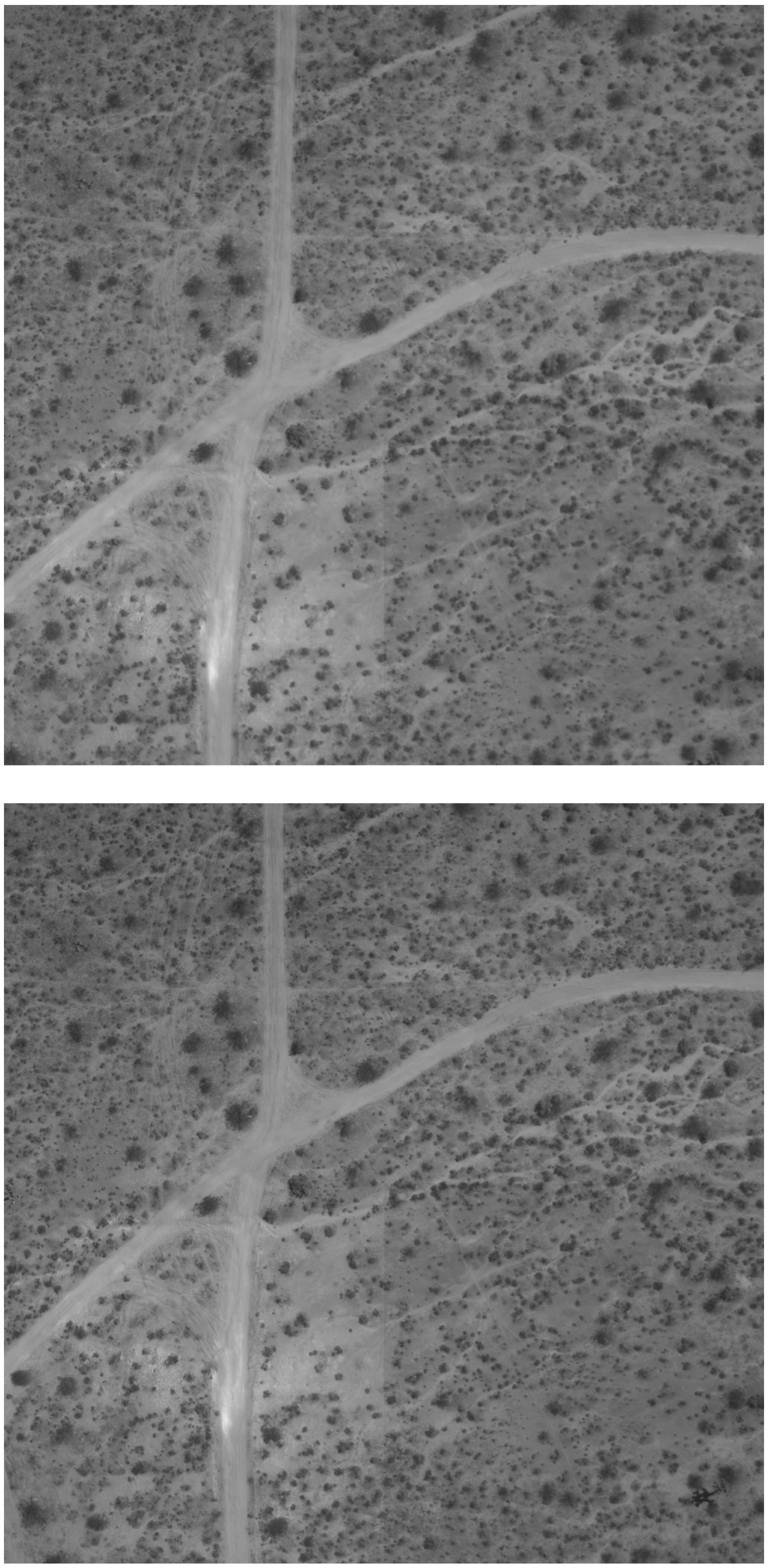


FIGURE 4.9: Image pair 2, prior to registration.

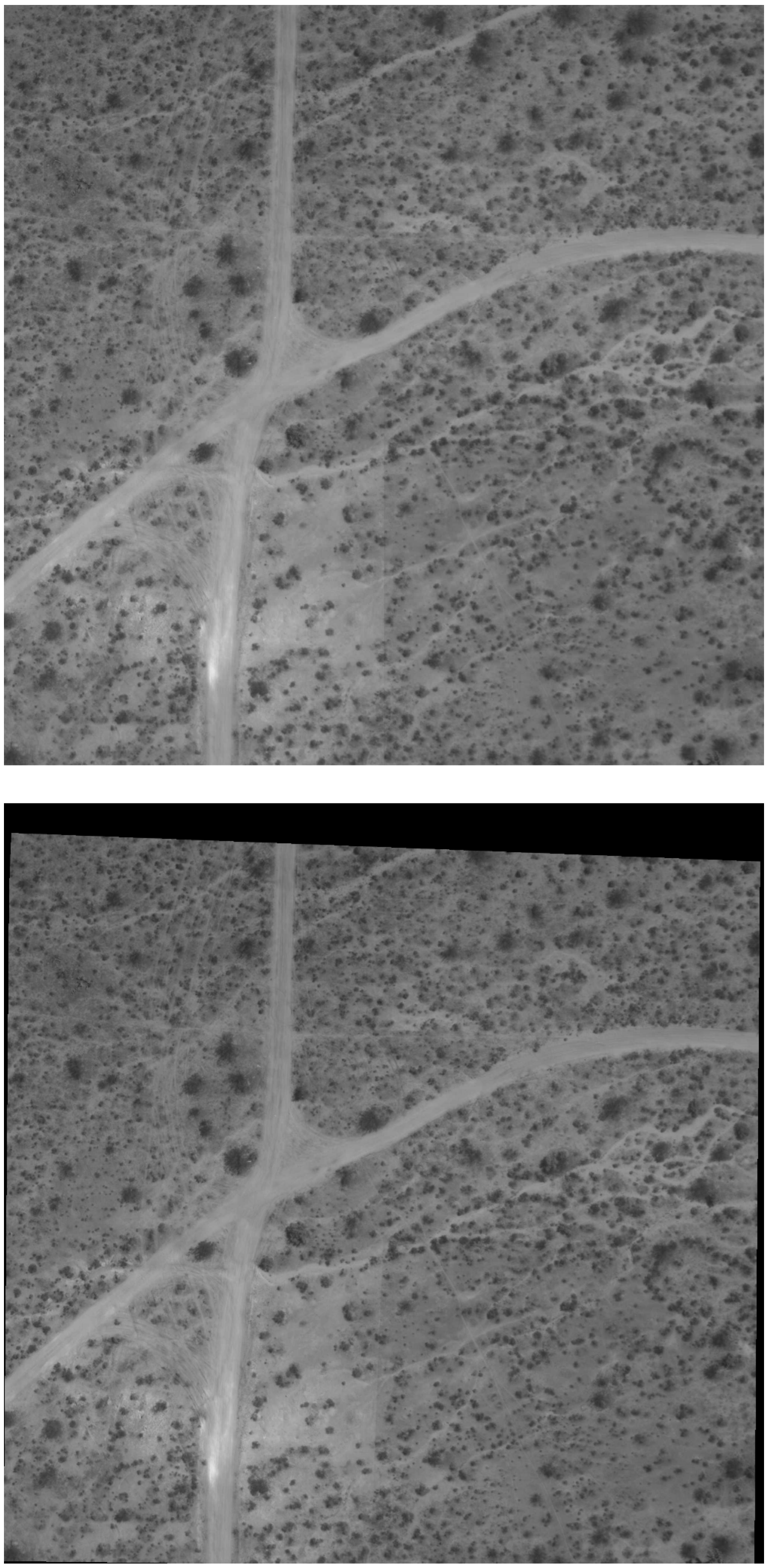


FIGURE 4.10: Image pair 2, after registration.

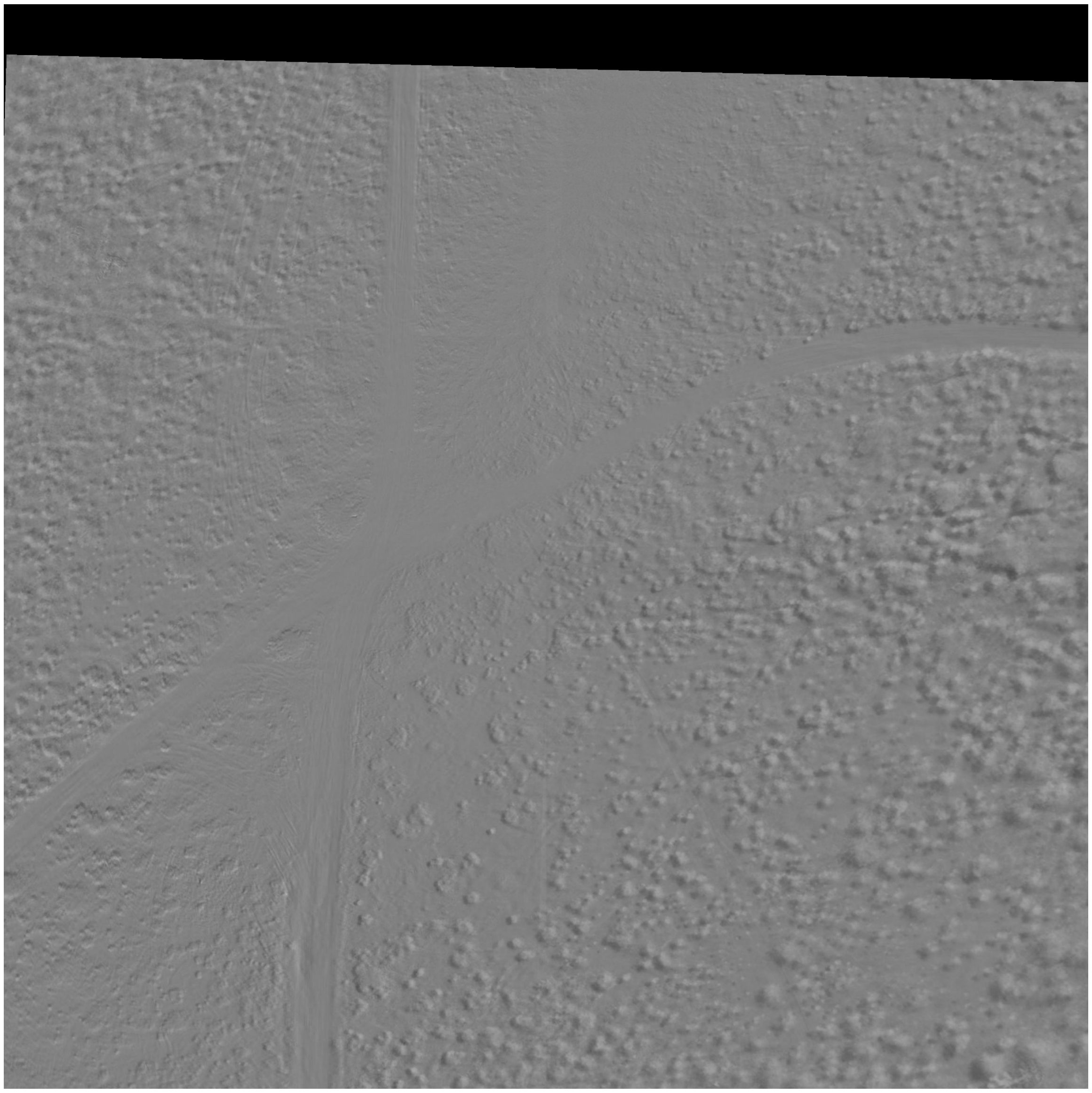


FIGURE 4.11: The difference image from pair 2, after initial alignment.

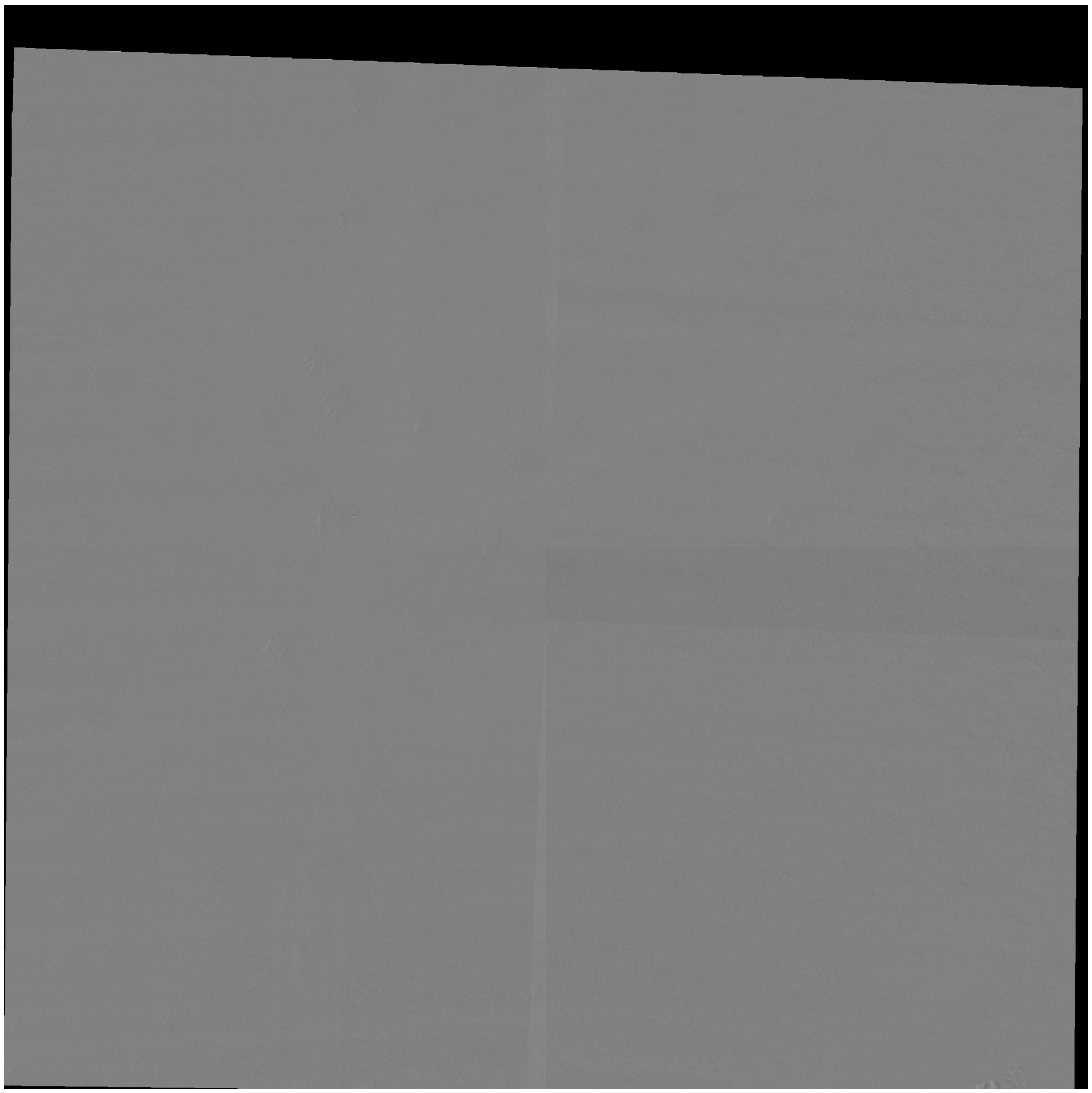


FIGURE 4.12: The difference image from pair 2, after correlated alignment.

base	warp	warped	dist0	dist1
$(128, 256)$	$(116, 268)$	$(128, 255)$	17.0	1.0
$(256, 256)$	$(245, 267)$	$(255, 256)$	15.6	1.0
$(384, 256)$	$(375, 265)$	$(384, 255)$	12.7	1.0
$(512, 256)$	$(505, 264)$	$(512, 256)$	10.6	0.0
$(640, 256)$	$(634, 262)$	$(639, 255)$	8.5	1.4
$(768, 256)$	$(764, 260)$	$(768, 255)$	5.7	1.0
$(896, 256)$	$(894, 259)$	$(896, 255)$	3.6	1.0
$(1024, 256)$	$(1023, 257)$	$(1024, 255)$	1.4	1.0
$(1152, 256)$	$(1152, 256)$	$(1151, 256)$	0.0	1.0
$(1280, 256)$	$(1282, 254)$	$(1280, 255)$	2.8	1.0
$(1408, 256)$	$(1411, 253)$	$(1407, 256)$	4.2	1.0
$(1536, 256)$	$(1541, 251)$	$(1535, 255)$	7.1	1.4
$(1664, 256)$	$(1671, 249)$	$(1664, 255)$	9.9	1.0
$(1792, 256)$	$(1800, 248)$	$(1791, 255)$	11.3	1.4
$(128, 384)$	$(117, 396)$	$(128, 384)$	16.3	0.0
$(256, 384)$	$(246, 395)$	$(255, 384)$	14.9	1.0
$(384, 384)$	$(376, 393)$	$(383, 384)$	12.0	1.0
$(512, 384)$	$(506, 391)$	$(512, 383)$	9.2	1.0
$(512, 384)$	$(506, 391)$	$(512, 383)$	9.2	1.0

TABLE 4.3: A portion of the control point migration table from pair 2.

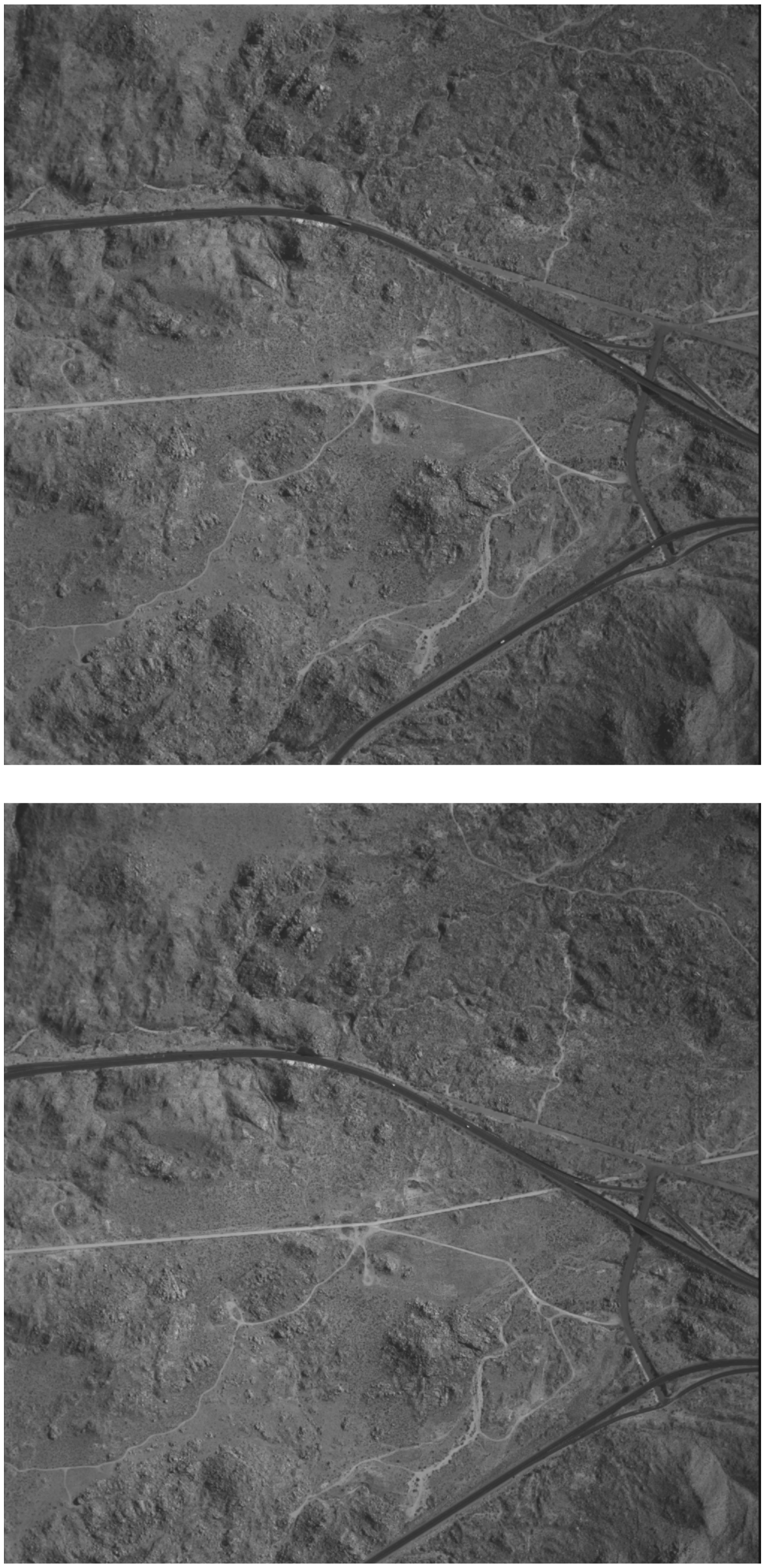
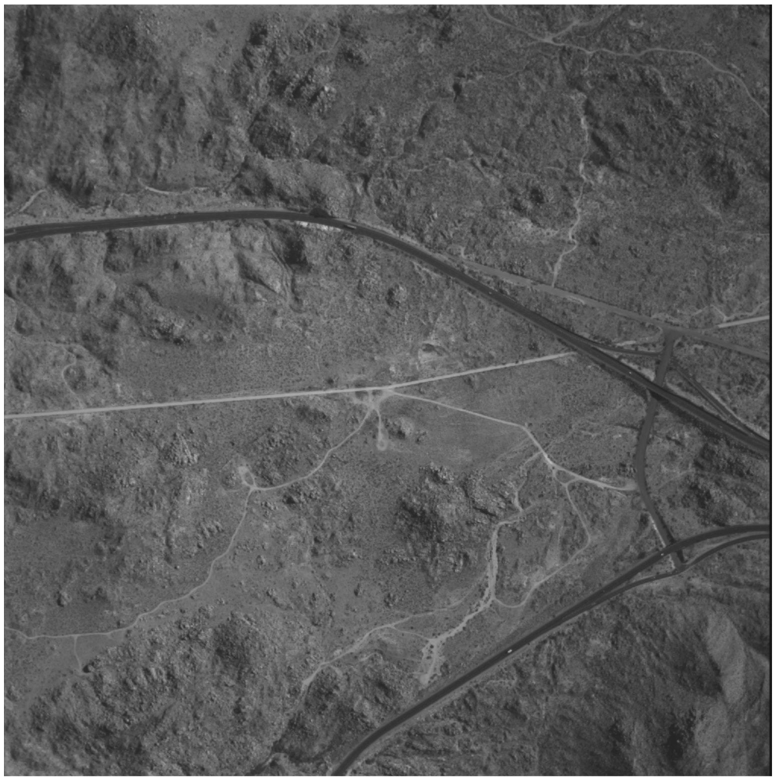


FIGURE 4.13: Image pair 3, prior to registration.



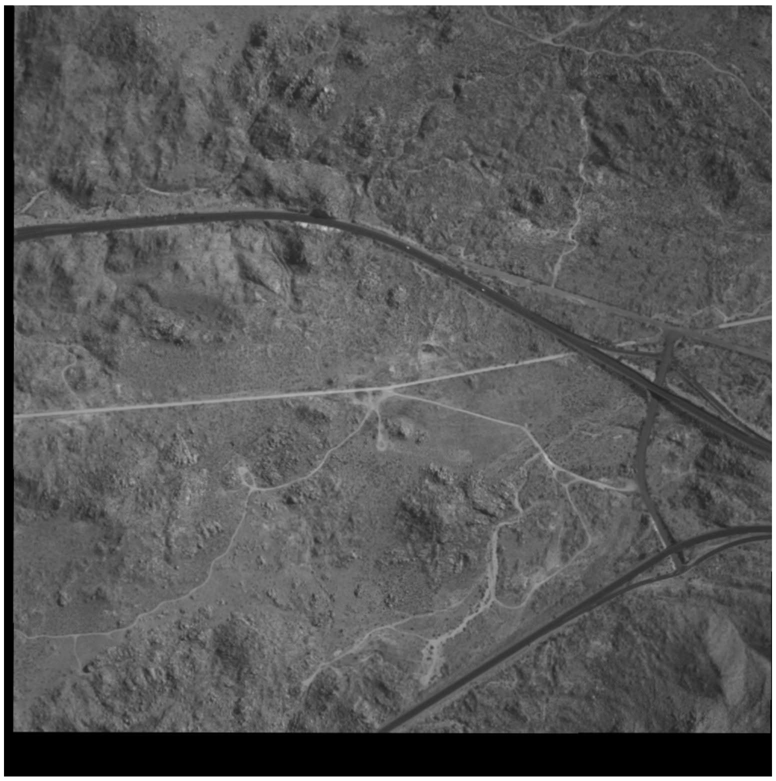


FIGURE 4.14: Image pair 3, after registration.

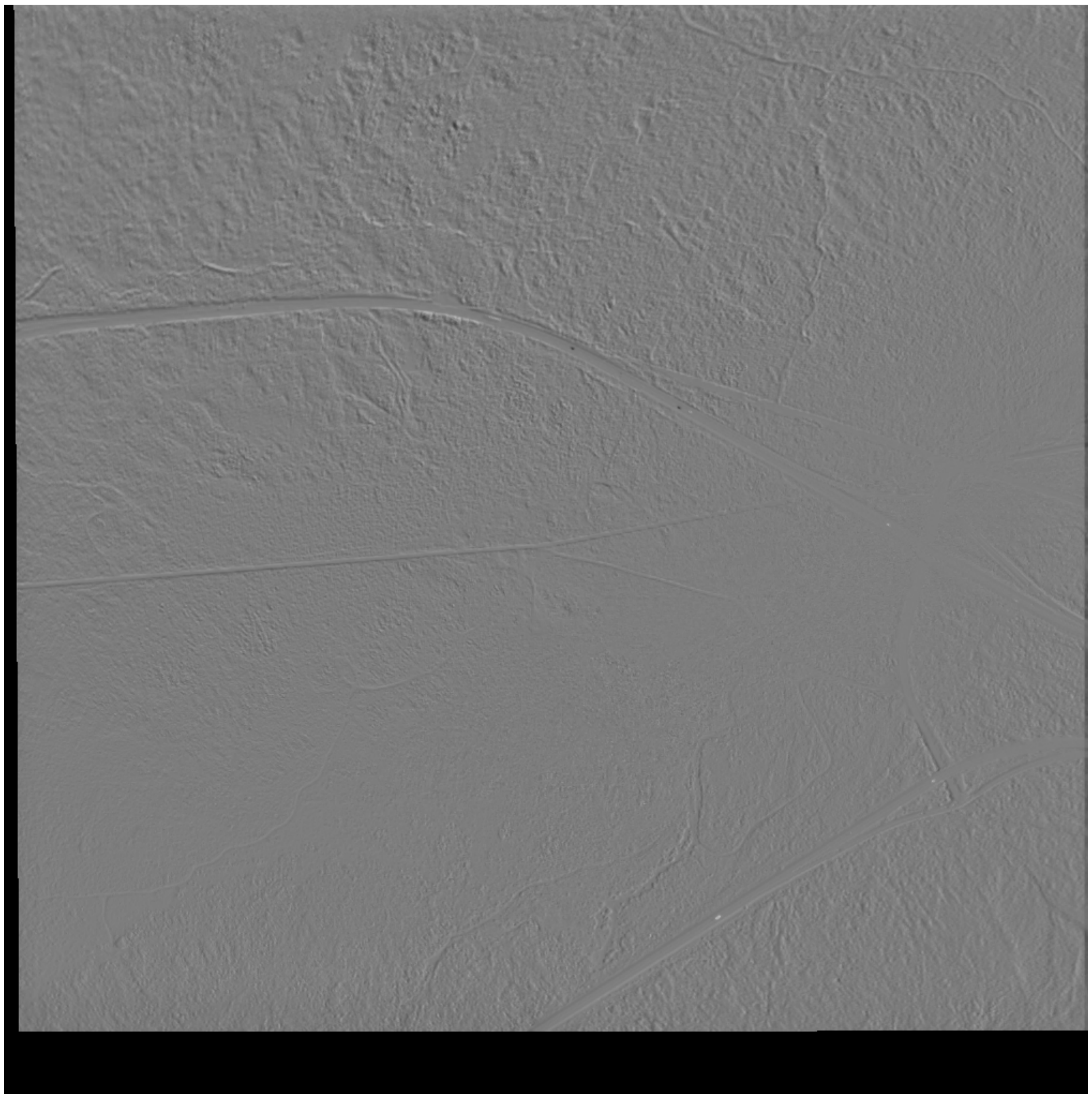


FIGURE 4.15: The difference image from pair 3, after initial alignment.

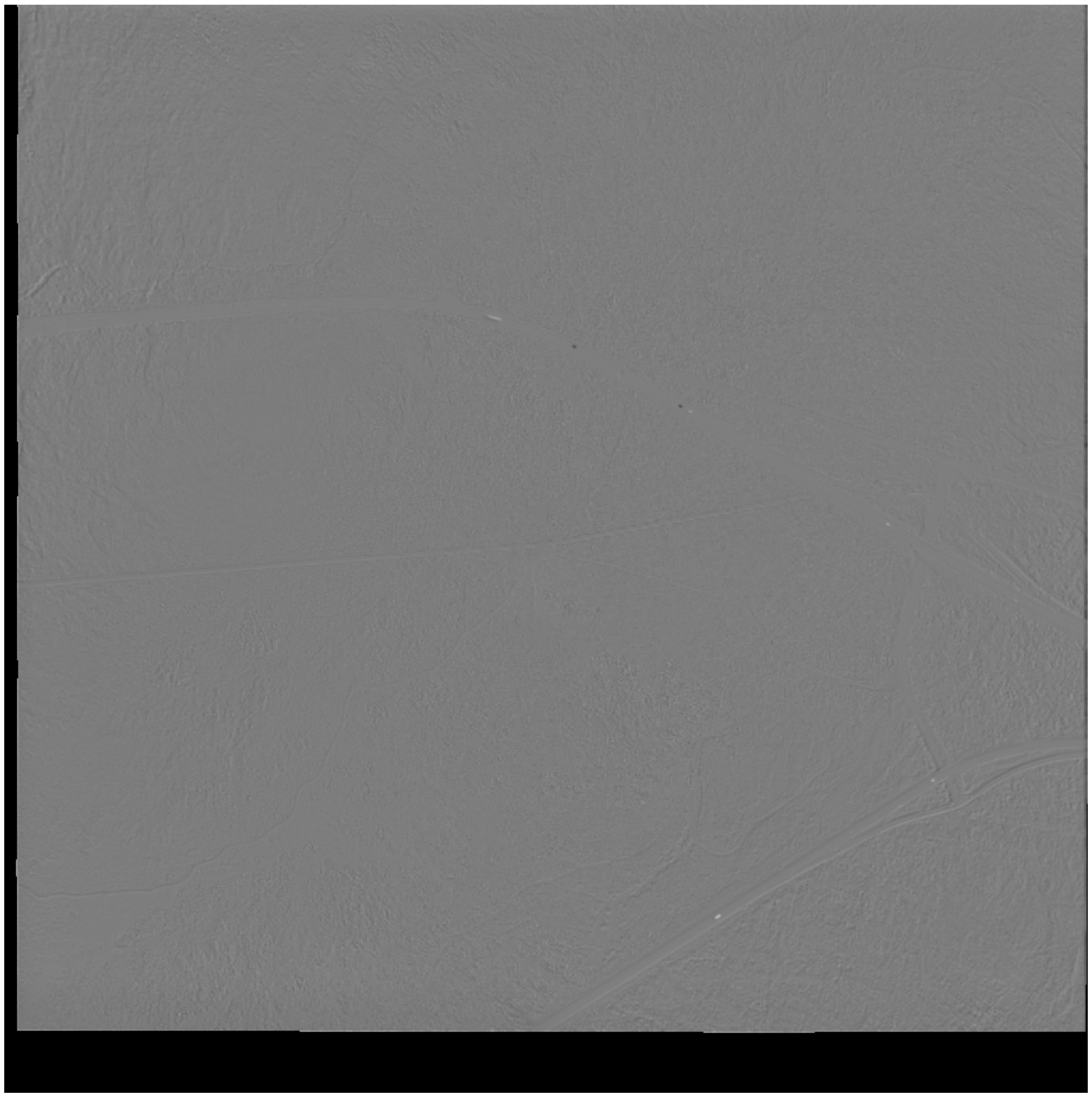


FIGURE 4.16: The difference image from pair 3, after correlated alignment.

base	warp	warped	dist0	dist1
$(128, 128)$	$(126, 132)$	$(128, 128)$	4.5	0.0
$(256, 128)$	$(253, 131)$	$(255, 128)$	4.2	1.0
$(384, 128)$	$(381, 131)$	$(383, 128)$	4.2	1.0
$(512, 128)$	$(509, 130)$	$(511, 127)$	3.6	1.4
$(640, 128)$	$(637, 130)$	$(639, 128)$	3.6	1.0
$(768, 128)$	$(765, 129)$	$(767, 127)$	3.2	1.4
$(896, 128)$	$(893, 129)$	$(895, 127)$	3.2	1.4
$(128, 256)$	$(127, 259)$	$(129, 256)$	3.2	1.0
$(256, 256)$	$(254, 259)$	$(256, 256)$	3.6	0.0
$(384, 256)$	$(382, 258)$	$(384, 255)$	2.8	1.0
$(512, 256)$	$(510, 258)$	$(512, 256)$	2.8	0.0
$(640, 256)$	$(638, 257)$	$(640, 255)$	2.2	1.0
$(768, 256)$	$(766, 257)$	$(767, 255)$	2.2	1.4
$(896, 256)$	$(894, 256)$	$(895, 255)$	2.0	1.4
$(128, 384)$	$(127, 386)$	$(128, 383)$	2.2	1.0
$(256, 384)$	$(255, 386)$	$(256, 384)$	2.2	0.0
$(384, 384)$	$(382, 386)$	$(383, 384)$	2.8	1.0
$(512, 384)$	$(510, 385)$	$(511, 383)$	2.2	1.4
$(512, 384)$	$(510, 385)$	$(511, 383)$	2.2	1.4

TABLE 4.4: A portion of the control point migration table from pair 3.

4.4 Conclusion

The image registration algorithm presented here has, in practice, shown itself to be very robust and adaptable to widely varying sensors and scenes. The remaining appendices present a detailed description of the implementation, as well as the source code itself.

BIBLIOGRAPHY

[1] Adobe Systems. *TIFF Revision 6.0*, 1992. The official specification for Tagged Image File Format (TIFF). TIFF is very popular in the remote sensing community due to its flexibility and extensibility. It supports a wide number of exotic pixel datatypes (including floating point values), and has a rich set of optional metadata fields which can be stored in the file header.

[2] David A. Forsyth and Jean Ponce. *Computer Vision: A Modern Approach.* Prentice Hall, 2003.

[3] Rafael C. Gonzalez and Richard E. Woods. *Digital Image Processing.* Addison-Wesley, 1992.

[4] Mohinder S. Grewal, Lawrence R. Weill, and Angus P. Andrews. *Global Positioning Systems, Interial Navigation, and Integration.* John Wiley & Sons, 2001.

[5] Charles V. Jakowatz, Jr., Daniel E. Wahl, Paul H. Eichel, Dennis C. Ghiglia, and Paul A. Thompson. *Spotlight-Mode Synthetic Aperture Radar: A Signal Processing Approach.* Kluwer Academic Publishers, 1996.

[6] Elliot D. Kaplan, editor. *Understanding GPS: Principles and Applications.* Artech House, 1996.

[7] Jon C. Leachtenauer and Ronald G. Driggers. *Surveillance and Reconnaissance Imaging Systems: Modeling and Performance Prediction.* Artech House, 2001. An excellent, high-level overview of the military and intelligence aspects of airborne imaging systems.

[8] Franz W. Leberl. *Radargrammetric Image Processing.* Artech House, 1990. Though currently out of print, this book is widely considered to be the classic text on SAR image processing applications. Most books on SAR image processing focus on image formation algorithms, whereas this book is more focused on what one can do with SAR images once they've been acquired.

[9] Edward M. Mikhail, James S. Bethel, and J. Chris McGlone. *Introduction to Modern Photogrammetry.* John Wiley & Sons, 2001.

[10] Bradford W. Parkinson and James J. Spilker, Jr., editors. *Global Positioning System: Theory and Applications*, volume one. American Institute of Astronautics and Aeronautics, 1996.

[11] Bradford W. Parkinson and James J. Spilker, Jr., editors. *Global Positioning System: Theory and Applications*, volume two. American Institute of Astronautics and Aeronautics, 1996.

[12] Niles Ritter and Mike Ruth. *GeoTIFF Format Specification*, 1995. The official specification for GeoTIFF, a set of geospatial extensions to the popular

TIFF image file format. GeoTIFF provides the ability to embed geolocation and geocoding metadata in the TIFF file header, and supports every major geospatial coordinate system in use today.

[13] Robert A. Schowengerdt. *Remote Sensing: Models and Methods for Image Processing.* Academic Press, second edition, 1997.

[14] Steven W. Smith. *Digital Signal Processing: A Practical Guide for Engineers and Scientists.* Newnes, 2003.

[15] John P. Snyder. Map projections - a working manual. Professional Paper 1395, United States Geological Survey, 1987.

[16] Milan Sonka, Vaclav Hlavac, and Roger Boyle. *Image Processing, Analysis, and Machine Vision.* PWS, second edition, 1999.

[17] United States Defense Mapping Agency. *DMATM-8358.2*, 1989. The official specication for the Universal Transverse Mercator (UTM) and Universal Polar Stereographic (UPS) geospatial coordinate systems.

[18] United States Department of Defense. *MIL-PRF-89020A*, 1996. The official specification for precision and storage of digital terrain elevation data (DTED).

[19] United States Department of Defense. *MIL-STD-2500B*, 1997. The official specification for version 2.1 of the National Imagery Transmission Format (NITF). This is the standard file format for imagery within the intelligence community. NITF is extremely flexible, and allows a great deal of metadata to be stored with the image data.

[20] United States National Imagery and Mapping Agency. *STDI-0002*, 2000. The official compendium of standardized extensions to the NITF 2.1 specification. Highly specialized sets of NITF extensions are available, including sensor-specific extensions for airborne and satellite imaging. This standard documents all extensions which have been officially endorsed by the NITF technical board.

APPENDIX A: IMPLEMENTATION DETAILS

The software was written in the Python programming language, using several third-party libraries for scientific computing and image processing. Python is a very high-level, object-oriented interpreted language. It is most commonly used for text processing, scripting, and web development, however it is a general purpose language that is easily adaptable to many other domains. Python is generally placed into the same category of languages as Perl, Tcl, and Ruby.

The major impediment to using interpreted languages for *HPC* (high-performance computing) applications, such as image processing, has been performance. The core of most HPC applications are operations on thousands (or millions, or billions) of numeric datatypes which are directly implemented in the hardware of most processors. Such datatypes include 8-bit, 16-bit, and 32-bit integers, both signed and unsigned. IEEE 754 floating-point values (single and double-precision) are also heavily used.

When using a compiled language such as C or Fortran, numeric operations in the source code are directly translated to machine operations on the CPU's native datatypes in the object code. In interpreted languages such as Python, however, there is an additional layer of abstraction. An interpreted language will typically have a high-level datatype as a wrapper around each native datatype. Each numeric operation incurs one or more function calls as overhead, as the interpreter transitions from the high-level wrapper to the native datatype and back again. At the very least, accessing variables in interpreted languages requires the interpreter to search through a dynamic symbol table to locate a pointer to an instance of a native datatype in memory.

For example, the following C function converts an array of values from degrees to radians:

```
void deg2rad(double* vals, int n)
{
    int i;

    for(i = 0; i < n; i++)
        vals[i] *= M_PI / 180;
}
```

The code is extremely efficient; it uses simple pointer arithmetic to iterate through the array, and performs in-place multiplication on its contents. Furthermore, any decent compiler will evaluate `M_PI / 180` once at compile time, rather than generating code that evaluates the expression at run time, every time around the loop.

Contrast this with the equivalent Python code:

```
function deg2rad(vals):
   for i in xrange(len(vals)):
       vals[i] *= math.pi / 180
```

On the surface, the code looks very similar, however it is not nearly as efficient. Each time around the loop, an element is extracted from the list `vals`. Lists are the Python equivalent to arrays, but the elements in a given list do not have to all be the same datatype. Therefore, after indexing into the list, the interpreter has to determine the type of that element. It then has to determine the meaning of that type when used as an operand to `*=`, with a floating-point value as the other operand. If the operation is legal, it dereferences a pointer to obtain the current value of the list element, computes the result of the expression, and stores the result back in memory.

The solution to this problem is to introduce a datatype in the interpreted language which encapsulates an array in the native environment. Furthermore, the standard operators and numeric functions are overloaded so that they can operate on these arrays in addition to scalar values. This would allow the code to be rewritten like so:

```
function deg2rad(vals):
    vals *= math.pi / 180
```

There is still more overhead than the C version, but the overhead is minimized and fixed; it does not increase with the size of the array. The transition between the interpreter's level of abstraction and native computations occurs once for the entire array, rather than once for each element of the array. In fact, the result may be faster than the C version, because the interpreter's implementation of `*=` may utilize SIMD CPU instructions such as the Pentium IV's SSE3 instruction set or the PowerPC's Altivec instruction set. These CPU instructions can perform a single arithmetic operation on multiple inputs simultaneously, rather than iterating through each input and performing the operation sequentially. Some C compilers are capable of recognizing optimization opportunities and generating SIMD-based code automatically, but this feature is almost always turned off by default and inconsistent when it is enabled.

This technique of introducing an efficient array datatype and parallelizing common numeric operations is quite popular. MATLAB is arguably the single most popular interpreted language for science and engineering research and simulations, and it uses such parallelization extensively. MATLAB's core datatype is a matrix; a scalar value is actually represented as a 1x1 matrix. All of the standard operators and nearly all of the core math functions can operate on matrices just as easily as scalars. The mark of a neophyte MATLAB programmer is writing code that iterates through an array, performing the same operation on each element, rather than performing the operation once on the entire array. In MATLAB jargon, eliminating iteration in favor of array operations is known as "vectorization".

This aspect of MATLAB's language design stems from the fact that it was originally written as an interactive teaching tool, to introduce students at the University of New Mexico to LINPACK and EISPACK. LINPACK was a widely used public domain Fortran library which contained a number of linear algebra functions. EISPACK was another Fortran library which contained functions to compute the eigenvalues and eigenvectors of a number of different matrix types. Both packages have since been subsumed into the LAPACK library, which is very widely used to this day.

MATLAB allows students to interactively create matrices and call functions in

LINPACK and EISPACK without having to do any Fortran programming. Using matrices rather than scalars as the core datatype was a natural design decision given the purpose of the application (an interactive teaching tool, rather than a general-purpose programming language). However, this design decision had a number of positive repercussions. It paved the way for high-performance computing in high-level, interpreted, dynamic languages. It taught programmers to intuitively think in terms of parallelized operations on data. Finally, it provided a means of expressing complex algorithms on large datasets in a very clear, concise manner with a minimum of extraneous syntax and boilerplate code.

The lessons learned from MATLAB are being applied to other interpreted languages. There are a number of extension libraries which provide this functionality in Python. The `array` package in the Python Standard Library offers some of the required functionality, but not enough to be generally useful. Numerical Python (also known as Numeric or NumPy) is probably the most popular add-on, and it effectively transforms Python into a MATLAB-quality number-crunching machine. More recently, a package called Numarray has been developed by STSI (the Space Telescope Science Institute) for image processing applications on data from the Hubble Space Telescope. Numarray was initially a reimplementation of Numeric with better performance characteristics for larger arrays, but it has since acquired a number of useful add-on modules which are not present in Numeric. The code presented here was written using Numarray, but much of it should be portable to Numeric.

APPENDIX B: SOURCE CODE

```
######################################################################
#
# File: runreg.py
#
# Description: contains main(), the master function for the
#   registration process.  PIL (Python Imaging Library) and numarray
#   are required in order to run the software.
#
######################################################################

import Image
import math
import numarray
import numarray.linear_algebra as la
import numarray.nd_image as img
import imgconv
import analysis
import registration as reg
import display
import sys
import bayer

def prep_xform(xform):
    newxform = numarray.zeros((2,2), numarray.Float64)
    invxform = la.inverse(xform)
    newxform[0, 0] = invxform[1, 1]
    newxform[0, 1] = invxform[1, 0]
    newxform[1, 0] = invxform[0, 1]
    newxform[1, 1] = invxform[0, 0]
    xoffset = invxform[0, 2]
    yoffset = invxform[1, 2]
    return (newxform, (yoffset, xoffset))

def analyze_xform(basepoints, warppoints, xform):
    a = numarray.array([0, 0, 1], type = numarray.Float64)
    delta = []
    tot0 = 0
    tot1 = 0
    logfile = open('point_migration.csv', 'wt')
    logfile.write('base x, base y, warp x, warp y, warped x, ' \
        'warped y, dist0, dist1\n')
    for i in xrange(len(basepoints)):
        base = basepoints[i]
        warp = warppoints[i]
        dist0 = math.sqrt((warp[0] - base[0])**2 \
            + (warp[1] - base[1])**2)
        a[0] = warp[0]
        a[1] = warp[1]
        warped = numarray.matrixmultiply(xform, a)
        dist1 = math.sqrt((warped[0] - base[0])**2 \
            + (warped[1] - base[1])**2)
        report = '%d, %d, %d, %d, %d, %d, %f, %f\n' % \
            (base[0], base[1], warp[0], \
            warp[1], warped[0], warped[1], \
            dist0, dist1)
        logfile.write(report)
        tot0 += dist0
        tot1 += dist1

    logfile.close()
    print 'tot0 = ', tot0
    print 'tot1 = ', tot1

def main():
    # Basic configuration
    basedir = '.'
    indir = 'data\\registration\\pair1'
    cfgfile = 'pair1.py'
    cfgpath = basedir + '\\' + indir

    # Parse the config file
    cfg = open(cfgpath + '\\' + cfgfile, 'rt')
    exec cfg
    cfg.close()
    print 'basefile =', basefile
    print 'warpfile =', warpfile

    # Generate the optimal affine transform
    xform = reg.genxform(basepts, warppts)
    print xform
    # analyze_xform(basepts, warppts, xform)

    # Read the input images and convert them to arrays
    im0 = Image.open(cfgpath + '\\' + basefile)
    im1 = Image.open(cfgpath + '\\' + warpfile)
    a0 = imgconv.image2array(im0)
```

```
a1 = imgconv.image2array(im1)

#a0 = a0[:, :, 0]
#a1 = a1[:, :, 0]

if bayerimg:
    # Perform Bayer decoding, convert to YUV,
    # then use the Y values.
    display.saveimage(a0, 'base_bayer.tif')
    rgb = bayer.decode(a0, 0.9)
    display.saveimage(rgb, 'base_rgb.tif')
    rgb = imgconv.normalize(rgb)
    rgb = imgconv.rgb2yuv(rgb)
    rgb *= 255
    a0 = rgb[..., ..., 0].astype(numarray.UInt8)

    display.saveimage(a1, 'warp_bayer.tif')
    rgb = bayer.decode(a1, 0.9)
    display.saveimage(rgb, 'warp_rgb.tif')
    rgb = imgconv.normalize(rgb)
    rgb = imgconv.rgb2yuv(rgb)
    rgb *= 255
    a1 = rgb[..., ..., 0].astype(numarray.UInt8)

display.saveimage(a0, 'base.tif')
display.saveimage(a1, 'warp0.tif')

# Initialize the warp image and save the output
affine = prep_xform(xform)
a2 = img.affine_transform(a1, affine[0], affine[1], order = 1)
print 'base min/max:', a1.min(), '/', a1.max()
print 'warp min/max:', a2.min(), '/', a2.max()
display.saveimage(a2, 'warp1.tif')

# Generate the disparity map
dmap = reg.mkdmap(a0, a2, 256, analysis.peak2rms)
cutoff = 1.0
dmapfilt = reg.filter_dmap(dmap, cutoff)
print 'Total number of disparities:', len(dmap)
print 'Disparities remaining after filter (cutoff = %f): %d' \
    % (cutoff, len(dmapfilt))
basepoints = []
warppoints = []

for i in dmapfilt:
```

```
            (xtrans, ytrans) = reg.gettrans(i[0])
            basex = i[3]
            basey = i[4]
            warpx = i[3] - xtrans
            warpy = i[4] - ytrans
            basepoints.append((basex, basey))
            warppoints.append((warpx, warpy))
            analysis.vizdisp(i)
            #display.surfplot(i[0])

        xform = reg.genxform(basepoints, warppoints)
        analyze_xform(basepoints, warppoints, xform)
        affine = prep_xform(xform)
        a3 = img.affine_transform(a2, affine[0], affine[1], order = 1)
        display.saveimage(a3, 'warp2.tif')

        # Create difference images for evaluation
        diff0 = analysis.diffimg(a0, a1)
        print '(diff0) min:', diff0.min()
        print '(diff0) max:', diff0.max()
        print '(diff0) stddev:', diff0.stddev()
        diffimg = imgconv.array2image(diff0)
        diffimg.save('diff0.tif')

        diff1 = analysis.diffimg(a0, a2)
        print '(diff1) min:', diff1.min()
        print '(diff1) max:', diff1.max()
        print '(diff1) stddev:', diff1.stddev()
        diffimg = imgconv.array2image(diff1)
        diffimg.save('diff1.tif')

        diff2 = analysis.diffimg(a0, a3)
        print '(diff2) min:', diff2.min()
        print '(diff2) max:', diff2.max()
        print '(diff2) stddev:', diff2.stddev()
        diffimg = imgconv.array2image(diff2)
        diffimg.save('diff2.tif')

    if __name__ == '__main__':
        main()
```

```
######################################################################
#
# File: analysis.py
#
# Description: contains functions used for automated data analysis.
#
######################################################################

import numarray
import numarray.nd_image as img
import math
import os
import imgconv
import display
import registration as reg

def peak2rms(a):
    '''Computes the peak:rms ratio of an array.'''
    peak = a.max()
    rms = math.sqrt((a**2).mean())
    return peak / rms

def diffimg(a, b):
    '''Generates a difference image between a and b.  The returned
    array is an 8-bit grayscale image.  Pixels which were black in
    either input are black in the difference image as well; they
    are assumed to be the result of a transform which left portions
    of an image without valid pixel data.  Both input images should
    be 2D Float32 or Float64 arrays.'''
    c = a.astype(numarray.Float64) - b.astype(numarray.Float64)
    c /= 255
    c /= 2
    c += 0.5
    c *= 255
    c = c.astype(numarray.UInt8)
    c[numarray.abs(a) < 1e-6] = 0
    c[numarray.abs(b) < 1e-6] = 0
    return c

def vizdisp(node):
    '''Visualize the result of processing a single node of a disparity
    map.  A subdirectory is created for the output.  The name of the
    subdirectory is node_xxxx_yyyy, in which the xxxx and the yyyy
    refer to the pixel coordinates of the center of the tiles used
    to generate the correlation surface.  Within this directory, the
```

```
following output is generated: the extracted tiles from the two
images, the correlation surface, a copy of the first tile with
crosshairs superimposed over the center, and a translated version
of the second tile with similar crosshairs.'''
try:
    os.mkdir('output')
except OSError:
    pass

os.chdir('output')
dname = 'node_%04d_%04d' % (node[3], node[4])

try:
    os.mkdir(dname)
except OSError:
    pass

os.chdir(dname)

# Generate 8-bit versions of the base and warp tiles
base = node[1].astype(numarray.UInt8)
warp = node[2].astype(numarray.UInt8)

# Generate an RGB image from the base tile, superimposing red
# crosshairs over the center.
shape3 = (base.shape[0], base.shape[1], 3)
rgb = numarray.array(shape = shape3, type = numarray.UInt8)
rgb[..., 0] = base
rgb[..., 1] = base
rgb[..., 2] = base
halfx = base.shape[1] / 2
halfy = base.shape[0] / 2
rgb[halfy, :, 0] = 255
rgb[:, halfx, 0] = 255
im = imgconv.array2image(rgb)
im.save('base.tif')

# Generate an RGB image from the warp tile, superimposing red
# crosshairs over the center.
shape3 = (warp.shape[0], warp.shape[1], 3)
rgb = numarray.array(shape = shape3, type = numarray.UInt8)
rgb[..., 0] = warp
rgb[..., 1] = warp
rgb[..., 2] = warp
halfx = warp.shape[1] / 2
```

```
halfy = warp.shape[0] / 2
rgb[halfy, :, 0] = 255
rgb[:, halfx, 0] = 255
im = imgconv.array2image(rgb)
im.save('warp0.tif')

# Translate the warp tile, generate an RGB image from it,
# and superimpose red crosshairs over the new center.
(xtrans, ytrans) = reg.gettrans(node[0])
warped = img.shift(warp, (ytrans, xtrans), order = 1, \
    prefilter = False)
rgb[..., 0] = warped
rgb[..., 1] = warped
rgb[..., 2] = warped
halfx = warped.shape[1] / 2
halfy = warped.shape[0] / 2
rgb[halfy, :, 0] = 255
rgb[:, halfx, 0] = 255
im = imgconv.array2image(rgb)
im.save('warp1.tif')

# Write some useful information out to a log file
log = file('log.txt', 'wt')
log.write('x: %d\n' % xtrans)
log.write('y: %d\n' % ytrans)
log.close()
os.chdir('..\\..')
```

```
######################################################################
#
# File: bayer.py
#
# Description: contains functions used for Bayer image decoding.
#
######################################################################

import numarray
import Image
import imgconv
import time

def whitebalance(a):
    '''White balances the input, a Bayer-encoded image.  Uses the
    'gray world' model, which assumes that the mean value of all
    three colors should be the same.'''
    r = a[::2, 1::2]
    g0 = a[::2, ::2]
    g1 = a[1::2, 1::2]
    b = a[1::2, ::2]
    avgr = r.mean()
    avgg = (g0.mean() + g1.mean()) / 2
    avgb = b.mean()
    cr = 1.0
    cg = 1.0
    cb = 1.0

    if avgr > avgg and avgr > avgb:
        cg = avgr / avgg
        cb = avgr / avgb
    elif avgg > avgr and avgg > avgb:
        cr = avgg / avgr
        cb = avgg / avgb
    else:
        cr = avgb / avgr
        cg = avgb / avgg

    print 'White balancing coefficients: R/G/B = %.3f/%.3f/%.3f' \
        % (cr, cg, cb)
    r *= cr
    g0 *= cg
    g1 *= cg
    b *= cb
```

```
def interpg(a):
    '''Interpolates the G value for an RGB triplet using correlated
    Bayer interpolation.  The input should be a 5x5 numarray array.'''
    vert = numarray.abs(a[0, 2] - a[4, 2])
    horz = numarray.abs(a[2, 0] - a[2, 4])

    if vert < horz:
        return (a[1, 2] + a[3, 2]) / 2
    elif horz < vert:
        return (a[2, 1] + a[2, 3]) / 2
    else:
        return (a[1, 2] + a[3, 2] + a[2, 1] + a[2, 3]) / 4

def interp(a, x, y):
    '''Generates an RGB triplet for a single pixel using Bayer
    interpolation.  'a' is a numarray array, and (y, x)
    is the pixel coordinate of interest.  Returns a tuple containing
    the interpolated RGB values.'''
    rgb = numarray.zeros(3, numarray.Float64)

    if y % 2:        # An odd row: BGBGBG...
        if x % 2:    # An odd column (green)
            rgb[0] = (a[y - 1, x] + a[y + 1, x]) / 2.0
            rgb[1] = a[y, x]
            rgb[2] = (a[y, x - 1] + a[y, x + 1]) / 2.0
        else:        # An even column (blue)
            rgb[0] = (a[y - 1, x - 1] + a[y - 1, x + 1] \
                + a[y + 1, x - 1] + a[y + 1, x + 1])\
                / 4.0
            rgb[1] = interpg(a[(y - 2):(y + 3), \
                (x - 2):(x + 3)])
            rgb[2] = a[y, x]
    else:            # An even row: GRGRGR...
        if x % 2:    # An odd column (red)
            rgb[0] = a[y, x]
            rgb[1] = interpg(a[(y - 2):(y + 3), \
                (x - 2):(x + 3)])
            rgb[2] = (a[y - 1, x - 1] + a[y - 1, x + 1] \
                + a[y + 1, x - 1] + a[y + 1, x + 1])\
                / 4.0
        else:        # An even column (green)
            rgb[0] = (a[y, x - 1] + a[y, x + 1]) / 2.0
            rgb[1] = a[y, x]
            rgb[2] = (a[y - 1, x] + a[y + 1, x]) / 2.0
    return rgb

def interp2(a, x, y):
    '''Generates an RGB triplet for a single pixel using Bayer
    interpolation.  'a' is a numarray array, and (y, x)
    is the pixel coordinate of interest.  Returns a tuple
    containing the interpolated RGB values.  Slightly different
    in implementation than interp(), and (surprisingly) slightly
    slower.'''
    win = a[(y - 2):(y + 3), (x - 2):(x + 3)]
    rgb = numarray.zeros(3, numarray.Float64)

    if y % 2:       # An odd row: BGBGBG...
        if x % 2:   # An odd column (green)
            rgb[0] = (win[1, 2] + win[3, 2])[0] / 2.0
            rgb[1] = win[2, 2][0]
            rgb[2] = (win[2, 1] + win[2, 3])[0] / 2.0
        else:       # An even column (blue)
            rgb[0] = (win[1, 1] + win[1, 3] + win[3, 1] \
                + win[3, 3])[0] / 4.0
            rgb[1] = interpg(win)
            rgb[2] = win[2, 2][0]
    else:           # An even row: GRGRGR...
        if x % 2:   # An odd column (red)
            rgb[0] = win[2, 2][0]
            rgb[1] = interpg(win)
            rgb[2] = (win[1, 1] + win[1, 3] + win[3, 1] \
                + win[3, 3])[0] / 4.0
        else:       # An even column (green)
            rgb[0] = (win[2, 1] + win[2, 3])[0] / 2.0
            rgb[1] = win[2, 2][0]
            rgb[2] = (win[1, 2] + win[3, 2])[0] / 2.0

    return rgb

def bayer(a, satmat):
    '''Converts a grayscale image to an RGB image using Bayer
    interpolation.  The function also does white balancing and
    color saturation on the image.'''
    b = numarray.zeros((a.shape[0] - 4, a.shape[1] - 4, 3), \
        numarray.Float64)

    for y in range(b.shape[0]):
        if y % 100 == 0:
            print 'Row %d/%d' % (y + 1, b.shape[0])
        for x in range(b.shape[1]):
            if satmat == None:
                b[y, x, :] = interp(a, x + 2, y + 2)
            else:
                b[y, x, :] = \
                    numarray.matrixmultiply( \
                    satmat, interp(a, x + 2, \
                    y + 2))

    return b

def adjrange(a):
    '''Adjusts the dynamic range of input image, an array of RGB
    triplets.  Sets the upper and lower bounds of the color
    components to [0..255].  Only has an effect on oversaturated
    images; if all components are already within the acceptable
    bounds, nothing is done.'''
    low = a.min()
    print 'Low:', low
    if low < 0.0:
        a -= low

    high = a.max()
    print 'High:', high
    if high > 255.0:
        range = high - low
        scale = 255 / range
        a *= scale

    # Clamp the values to ward off floating-point oddities
    lt0 = numarray.where(a < 0.0)
    gt255 = numarray.where(a > 255.0)
    print 'Clamping', lt0[0].nelements(), 'at 0'
    print 'Clamping', gt255[0].nelements(), 'at 255'
    a[lt0] = 0.0
    a[gt255] = 255.0

def decode(a, k = 1.0, bal = True):
    # Create the color saturation matrix
    satmat = None

    if k != 1.0:
        r0 = [0.299 + 0.701 * k, 0.587 * (1 - k), \
            0.114 * (1 - k)]
        r1 = [0.299 * (1 - k), 0.587 + 0.413 * k, \
            0.114 * (1 - k)]
        r2 = [0.299 * (1 - k), 0.587 * (1 - k), \
            0.114 + 0.886 * k]
        satmat = numarray.array([r0, r1, r2], \
            type = numarray.Float64)

    b = a.astype(numarray.Float64)

    if bal:
        whitebalance(b)

    c = bayer(b, satmat)
    adjrange(c)
    return c.astype(numarray.UInt8)
```

```
######################################################################
#
# File: display.py
#
# Description: contains functions for data visualization.
#
######################################################################

import numarray as na
import dislin
import imgconv

def surfplot(a):
    '''Displays a surface plot of a, a 2D array.'''
    xrng = na.arange(a.shape[0])
    yrng = na.arange(a.shape[1])
    dislin.surshade(a, xrng, yrng)
    dislin.disfin()

def saveimage(a, name):
    b = a.copy()
    im = imgconv.array2image(b.astype(na.UInt8))
    im.save(name)
```

```
######################################################################
#
# File: imgconv.py
#
# Description: contains functions to convert between image formats:
#   PIL/numarray, RGB/YUV, etc.
#
######################################################################

import Image
import numarray

def image2array(im):
    '''Converts a PIL Image object to a numarray array object.'''
    if im.mode == 'L' or im.mode == 'RGB':
        a = numarray.fromstring(im.tostring(), numarray.UInt8)
    elif im.mode == 'F':
        a = numarray.fromstring(im.tostring(), numarray.Float32)
    else:
        print 'mode:', im.mode
        raise ValueError, 'unsupported image mode'

    if im.mode == 'L' or im.mode == 'F':
        a.shape = (im.size[1], im.size[0])
    elif im.mode == 'RGB':
        a.shape = (im.size[1], im.size[0], 3)
    return a

def array2image(a):
    '''Converts a numarray array object to a PIL Image object.'''
    mode = ''

    if len(a.shape) == 3 and a.shape[2] == 3:
        if a.typecode() == numarray.UInt8:
            mode = 'RGB'
    elif len(a.shape) == 2 or a.shape[2] == 1:
        if a.typecode() == numarray.UInt8:
            mode = 'L'
        elif a.typecode() == numarray.Float32:
            mode = 'F'

    if mode == 'L' or mode == 'F':
        return Image.fromstring(mode, (a.shape[1], a.shape[0]), \
            a.tostring())
    elif mode == 'RGB':
        ar = a[:, :, 0]
        ag = a[:, :, 1]
        ab = a[:, :, 2]
        ir = Image.fromstring('L', (ar.shape[1], ar.shape[0]), \
            ar.tostring())
        ig = Image.fromstring('L', (ag.shape[1], ag.shape[0]), \
            ag.tostring())
        ib = Image.fromstring('L', (ab.shape[1], ab.shape[0]), \
            ab.tostring())
        im = Image.merge('RGB', [ir, ig, ib])
        return Image.merge('RGB', [ir, ig, ib])
    else:
        raise ValueError, 'unsupported image mode'

def normalize(a):
    '''Normalizes an image.'''
    if a.typecode() == numarray.UInt8:
        b = a.astype(numarray.Float64)
        b /= 255;
    else:
        raise ValueError, 'unsupported image mode'

    return b

def rescale(a, low, high):
    amin = a.min()
    amax = a.max()
    adelta = amax - amin
    bdelta = high - low
    b = a.astype(numarray.Float64)
    b -= amin
    b /= adelta
    b *= bdelta
    b += low
    return b

def fftshift(a):
    b = numarray.zeros(a.shape, a.type())
    halfy = a.shape[0] / 2
    halfx = a.shape[1] / 2
    mody = a.shape[0] % 2
    modx = a.shape[1] % 2
    b[:(halfy + mody), :(halfx + modx)] = a[halfy:, halfx:]
    b[:(halfy + mody), (halfx + modx):] = a[halfy:, :halfx]
    b[(halfy + mody):, (halfx + modx):] = a[:halfy, :halfx]
    b[(halfy + mody):, :(halfx + modx)] = a[:halfy, halfx:]
    return b

def rgb2yuv(a):
    '''Converts an RGB image to the YUV colorspace.'''
    if a.shape[2] != 3:
        raise ValueError, 'image must have 3 color planes'

    b = numarray.zeros(a.shape, numarray.Float64)
    b[..., 0] = a[..., 0] * 0.299 + a[..., 1] * 0.587 \
        + a[..., 2] * 0.114
    b[..., 1] = a[..., 0] * -0.147 + a[..., 1] * -0.289 \
        + a[..., 2] * 0.436
    b[..., 2] = a[..., 0] * 0.615 + a[..., 1] * -0.515 \
        + a[..., 2] * -0.100
    return b
```

```
######################################################################
#
# File: registration.py
#
# Description: contains the core image registration functions.
#
######################################################################

import numarray as na
import numarray.fft as fft
import numarray.linear_algebra as la
import imgconv

def xcorr2d(a, b):
    '''A non-normalized 2D cross-correlation in the frequency domain.
    Returns the resultant correlation surface.  a and b must have the
    same dimensions; the correlation surface will have the same
    dimensions as a and b.'''
    afreq = fft.fft2d(a)
    bfreq = fft.fft2d(b)
    xcorr = afreq * bfreq.conjugate()
    csurf = fft.inverse_fft2d(xcorr)
    csurf = na.abs(csurf)
    csurf = imgconv.fftshift(csurf)
    return csurf

def gettrans(a):
    '''The input is a correlation surface in which the low-frequency
    components have been shifted to the center.  The function finds
    the peak of the surface and returns its coordinates relative to
    the center.  This value can be interpreted as a translation to
    be applied to one of the two input images used to generate the
    correlation surface, which will optimally align them.'''
    (y, x) = na.where(a == a.max())
    y = y[0]
    x = x[0]
    y -= a.shape[0] % 2 + a.shape[0] / 2
    x -= a.shape[1] % 2 + a.shape[1] / 2
    return (x, y)

def croppair(x0, x1, trans):
    '''Given that x0 and x1 are a pair of images to be registered,
    and x1 has already been translated by (xtrans, ytrans), crop
    out the overlapping regions of both images and return them.'''
    y0 = x0.copy()
    y1 = x1.copy()
    xtrans = trans[0]
    ytrans = trans[1]

    if xtrans >= 0:
        if ytrans >= 0:
            y0 = y0[ytrans:, xtrans:]
            y1 = y1[ytrans:, xtrans:]
        else:
            y0 = y0[:ytrans, xtrans:]
            y1 = y1[:ytrans, xtrans:]
    else:
        if ytrans >= 0:
            y0 = y0[ytrans:, :xtrans]
            y1 = y1[ytrans:, :xtrans]
        else:
            y0 = y0[:ytrans, :xtrans]
            y1 = y1[:ytrans, :xtrans]

    return (y0, y1)

def mkdmap(x0, x1, winsize, fomfunc):
    '''Generates a disparity map between x0 and x1, using the given
    correlation window size.  Returns a list of tuples; each tuple
    contains a correlation surface, the two cropped images used to
    create the surface, the x and y coordinates corresponding to the
    center of the cropped images in the original pixel space, and a
    figure of merit generated by feeding the correlation surface to
    the function fomfunc.'''
    dmap = [];

    for row in range(winsize, x0.shape[0] + winsize / 2, \
        winsize / 2):
        ctry = row - winsize / 2
        print 'Processing row %d' % ctry

        for col in range(winsize, x0.shape[1] + winsize / 2, \
            winsize / 2):
            ctrx = col - winsize / 2
            y0 = x0[(row - winsize):row, (col - winsize):col]
            y1 = x1[(row - winsize):row, (col - winsize):col]
            csurf = xcorr2d(y0, y1)
            #fom = fomfunc(csurf)
            fom = 1.01
            node = (csurf, y0, y1, ctrx, ctry, fom)
            dmap.append(node)

    return dmap

def filter_dmap(oldlist, cutoff):
    '''Should be applied to the list returned by mkdmap().  Returns
    a list containing only the elements of the input list which have
    a figure of merit greater than or equal to the cutoff value.
    Other criteria are applied as well.'''
    newlist = []

    for i in oldlist:
        a = i[0]
        (y, x) = na.where(a == a.max())
        x = x[0]
        y = y[0]

        # If the peak is on a border row or column, it's wrong.
        if x == 0 or x == a.shape[1] - 1 or y == 0 \
            or y == a.shape[0] - 1:
            continue

        # If more than 20% of the pixels in the base or warp tile are
        # black, then the correlation is unreliable.
        base = i[1]
        warp = i[2]
        npix = warp.size()
        nvalid = warp.flat.nonzero()[0].size()
        pctvalid = float(nvalid) / npix

        if pctvalid < 0.95:
            continue

        if i[5] >= cutoff:
            newlist.append(i)

    return newlist

def genxform(basepts, warppts):
    '''Generates an optimal affine transform which warps one image
    into another image's pixel space.  The inputs are two lists of
    tuples.  Each tuple contains an (x, y) image coordinate.  basepts
    is a list of pixel coordinates in the first image (the base).
    warppts is a list of corresponding pixel coordinates in the
    second image (the warp image).  Both lists must be the same
```

```
length.  In addition, basepts[i] must correspond to warppts[i]
for all i.'''
rows = len(basepts)
assert(rows == len(warppts))
assert(rows >= 3)
a = na.array(shape = (rows, 3), type = na.Float64)
bx = na.array(shape = (rows, 1), type = na.Float64)
by = na.array(shape = (rows, 1), type = na.Float64)

for i in range(rows):
    a[i, 0] = warppts[i][0]
    a[i, 1] = warppts[i][1]
    a[i, 2] = 1
    bx[i, 0] = basepts[i][0]
    by[i, 0] = basepts[i][1]

xlsq = la.linear_least_squares(a, bx)
ylsq = la.linear_least_squares(a, by)
xvals = xlsq[0].flat
yvals = ylsq[0].flat

# Extract the scale, shear, and translation values for x and y.
# This code is here for clarity more than anything else.
xscale = xvals[0]
xshear = xvals[1]
xtrans = xvals[2]
yscale = yvals[1]
yshear = yvals[0]
ytrans = yvals[2]

xform = na.array(shape = (3, 3), type = na.Float64)
xform[0] = [xscale, xshear, xtrans]
xform[1] = [yshear, yscale, ytrans]
xform[2] = [0, 0, 1]
return xform
```