SANDIA REPORT
SAND2019-11786

Printed 2019

Sandia National Laboratories

Image Processing Algorithms for Tuning Quantum Devices and Nitrogen-Vacancy Imaging

Cara Monical, Phil Lewis, Abrielle Agron, Kurt Larson, Andy Mounce

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185
Livermore, California 94550

Issued by Sandia National Laboratories, operated for the United States Department of Energy by National Technology & Engineering Solutions of Sandia, LLC.

Available to DOE and DOE contractors from

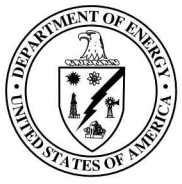
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@osti.gov
Online ordering: http://www.osti.gov/scitech

Available to the public from

U.S. Department of Commerce
National Technical Information Service
5301 Shawnee Road
Alexandria, VA 22312

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.gov
Online order: https://classic.ntis.gov/help/order-methods



ABSTRACT

Semiconductor quantum dot devices can be challenging to configure into a regime where they are suitable for qubit operation. This challenge arises from variations in gate control of quantum dot electron occupation and tunnel coupling between quantum dots on a single device or across several devices. Furthermore, a single control gate usually has capacitive coupling to multiple quantum dots and tunnel barriers between dots. If the device operator, be it human or machine, has quantitative knowledge of how gates control the electrostatic and dynamic properties of multiqubit devices, the operator can more quickly and easily navigate the multidimensional gate space to find a qubit operating regime.

We have developed and applied image analysis techniques to quantitatively detect where charge offsets from different quantum dots intersect, so called anticrossings. In this document we outline the details of our algorithm for detecting single anticrossings, which has been used to fine-tune the inter-dot tunnel rates for a three quantum dot system.[9] Additionally, we show that our algorithm can detect multiple anticrossings in the same dataset, which can aid in the coarse tuning the electron occupation of multiple quantum dots.

We also include an application of cross correlation to the imaging of magnetic fields using nitrogen vacancies.

The data for the automated detection of anticrossing has been provided by Jason Petta's group at Princeton University. The devices were fabricated in the Princeton University Quantum Device Nanofabrication Laboratory. Portions of this work were were included in reference [9] which describes an automated procedure for tuning a device using the algorithms described in this report. We thank Adam Mills and Mayer Feldman in Jason's group at Princeton for providing feedback on the algorithm and code as they incorporated the algorithm into their system. They pointed out data we were not yet handling well and provided suggestions that significantly improved the performance of the algorithm. The background to the problem and data for the imaging using nitrogen vacancy has been provided by Sandia Truman Fellow Pauli Kehayias.

CONTENTS

LIST OF FIGURES

1. AUTOMATED ANTICROSSING PARAMETER EXTRACTION

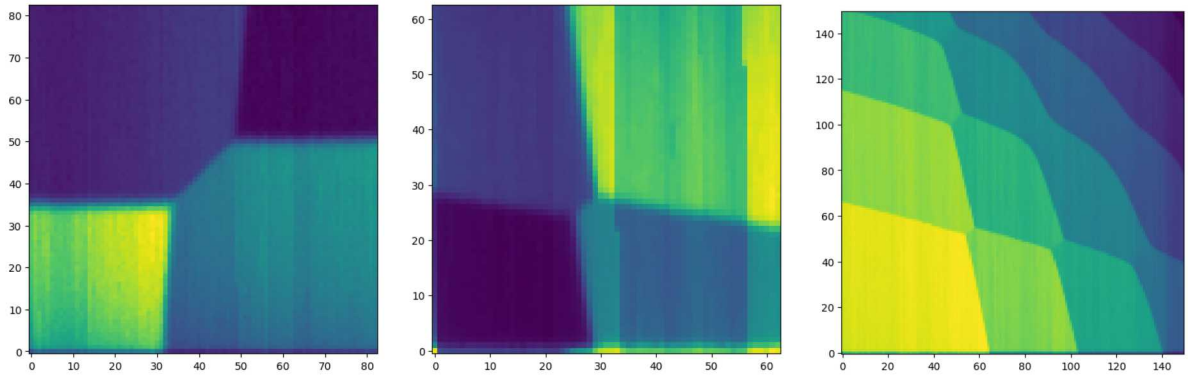
1.1. INTRODUCTION

Quantum computing can be said to have begun when R. Feynman introduced in the early 1980's the idea of a quantum computer that uses the principles of quantum mechanics to perform simulations that a classical computer can not [4]. In 1994, P. Shor developed a polynomial-time quantum algorithm for determining the prime factors of a number [11], a problem intractable on classical computers and the basis for many modern cryptographic schemes. Since then, there has been substantial research interest in quantum computing – both in building a quantum computer and in designing algorithms to run on one.

Like the bit for classical computers, the fundamental unit of quantum computing is the qubit. While there are several proposed architectures for the qubit, forming single qubit and two-qubit gates from semiconductor spin qubits seems to be a promising platform for achieving the scalability required to build a quantum computer capable of outperforming a classical computer on some problems [8]. Tuning a quantum dot for operation as a spin qubit requires complex hardware and software controls over the quantum mechanics of the system [10].

Extracting parameters from a charge-stability diagram allows for the creation of virtual gates which in turn allow for finer control of the quantum mechanics of the system [9]. An anticrossing in a charge-stability diagram consists of two triple points, each described by a point and two line segments that meet at the point, corresponding to electron transitions into or out of the quantum dot. The line segment connecting the two triple points may or may not be visible in the scan. We are interested in the locations of each triple point as well as the slopes of the line segments in the anticrossing. See figure 1-1 for examples of anticrossings in charge-stability diagrams.

As quantum devices get more complicated, automating their tuning is essential to making a scalable system. This report presents an algorithm that detects anticrossings in a charge-stability diagram and extracts their parameters using image processing techniques. The primary use case for this algorithm, and where the bulk of our development and testing has been centered, is in detecting a single anticrossing centered in the diagram as in figure 1-1(b). However, we can extend the algorithm to the case of multiple anticrossings as in fig 1-1(c) with just a single modification and a retuning of parameters. A description of how this algorithm is being used by J. Petta's group at Princeton is described in [9].



(a) A clean scan with a single anticrossing

(b) A more noisy scan with a single anticrossing

(c) A scan with multiple (at least 8) anticrossings

Figure 1-1. These are typical scans in the dataset used for this analysis.

1.1.1. Hough Transform

Our algorithms use the Hough line transform and its subsequent generalizations. After a 1962 patent by P. Hough [5], the commonly known form of the Hough line transform was introduced by R. Duda and P. Hart [3]. Because our algorithms rely fundamentally on the concepts introduced by Duda and Hart, we summarize their results here. The Hough line transform consists of first parameterizing a line L as the pair (θ, ρ) where θ is the angle of the normal of L and ρ is the distance between L and the origin. To detect lines in a binary image, we map points (x_i, y_i) to sinusoidal curves in the $\theta - \rho$ plane using

$$\rho = x_i \cos\theta + y_i \sin\theta.$$

Colinear points correspond to curves with a common point of intersection. Thus, we can detect lines by having every pixel vote for the points in the $\theta - \rho$ plane that form its corresponding sinusoidal curve. The points (θ, ρ) with the highest number of votes are the lines in the image with the most number of pixels on that line.

The Hough transform has been extended in a variety of ways, including, but not limited to, circles parameterized by (x, y, r) [3], a probabilistic method that determines the endpoints of the detected lines [7], shapes defined by analytical equations [1], and general shapes using convolutional templates [1]. We will apply the concepts of the Hough transform for analytical equations to the detection of anticrossings by defining an appropriate parameter space.

1.2. ALGORITHM

This section describes the main ideas for each step of the algorithm. Full details are included in the code snippets in appendix A. For a list of parameters in the algorithm and a discussion on how to set appropriate values see appendix B.

Before describing the algorithm, we must first establish some terminology and conventions. Our algorithm operates in pixel space and converts the final detected anticrossing to physical units using the

scan's parameters. Pixels are specified in Cartesian coordinates where $(0, 0)$ is the lower-left corner and (x, y) is the point x pixels to the right and y pixels up from the origin. Our algorithm is implemented in Python and uses SciPy for standard algorithms.

Given a line segment with slope m, we define the inclination of the line segment as the angle θ such that

$$m = \tan\theta \text{ and } -120° < \theta \leq 60°$$

where angles are measured in degrees from the horizontal in the counter-clockwise direction. We then parametrize an anticrossing as

$$A = (p_u, \theta_u, \phi_u, p_d, \theta_d, \phi_d)$$

where $p_u = (x_u, y_u)$ and $p_d = (x_d, y_d)$ are the pixel locations of the triple points and θ_u, ϕ_u, θ_d, and ϕ_d are the inclinations of the four line segments of A in order, starting with the upper left line segment and moving in a clockwise direction. Furthermore, let ψ be the inclination of the line segment (present or not) between p_u and p_d. See figure 1-2 for an illustration of these parameters.

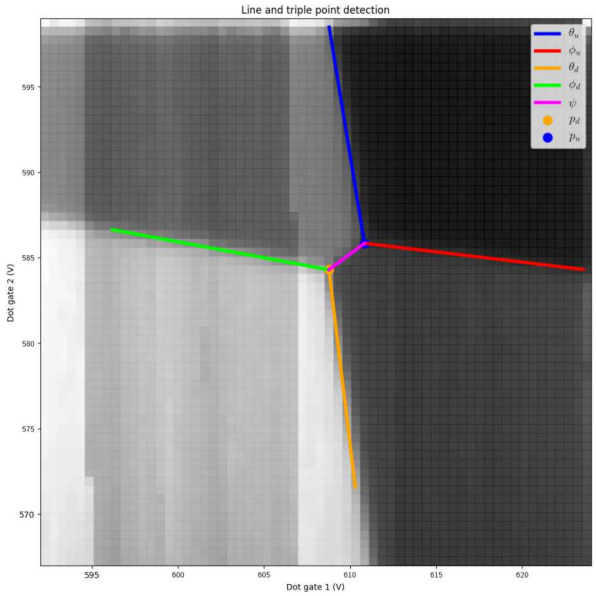


Figure 1-2. An anticrossing consists of two triple points p_u and p_d and four inclinations $\theta_u, \phi_u, \theta_d$, and ϕ_d as depicted above. The inclination ψ may or may not be visible in the scan and is determined from p_u and p_d.

Since we are expecting the anticrossing to be relatively centered in the charge-stability diagram, the analyst specifies a parameter M that defines what percentage of the image is the "middle". For example, for $M = 40$ percent, the valid region for the triple points is the region formed by removing the outer 30 percent from each edge of the scan. Valid anticrossings meet the following assumptions:

- p_u and p_d appear in the middle M percent of the image,
- $\phi_d > \theta_d$ and $\phi_u > \theta_u$,
- $\theta_d \approx \theta_u$ and $\phi_d \approx \phi_u$, and

- $\theta_d < \psi < \phi_d + 180°$.

Even accounting for these assumptions, searching the complete parameter space of valid anticrossings is impractical. Thus we perform a targeted search that finds a good approximation of the parameters, and then use these as the starting point of a more exhaustive, local search. This can be broken down to five main steps, each detailed in the following discussion:

1. convert charge-stability scan to a binary image of "white" and "black" pixels where white pixels represent the electron transitions (Section 1.2.1),
2. use the Hough line transform to determine possible inclinations of the four line segments of an anticrossing (Section 1.2.2),
3. detect possible triple points in binary image using Hough transform-like accumulators (Section 1.2.3),
4. use template matching to select most probable anticrossing(s) (Section 1.2.4), and
5. perform template-based local search to optimize the parameters of the detected anticrossing (Section 1.2.5).

1.2.1. Binary Image

The first step of our analysis is to convert the scan into a binary image where the white pixels correspond to the electron transitions of the charge-stability diagram. We do this by performing a few passes through the image – the first generates a candidate list of white pixels, while subsequent passes seek to improve the signal-to-noise ratio (SNR) of the binary image.

To start, we first linearly normalize the data to $[0, 1]$ and then loop through the rows of the data. For each row, we take the horizontal gradient and calculate a threshold of a specified percentile of the gradient. Anything below the threshold is set to 0. Ignoring pixels in the border of the data, we set peaks of at least a specified width as white while other pixels are set to black. We then repeat this process for each column and the vertical gradient, adding the peaks found to the set of white pixels.

After forming the initial set of white pixels, we perform various clean-up steps to remove some of the noise in the binary image. For the data used for this report, this consisted of removing switches (a noise pattern consisting of long vertical lines specific to the device), straightening lines, removing isolated pixels, and optionally performing binary closing. See figure 1-3 for the results of these steps. These steps have been tuned to the noise patterns specific to our data, while different devices may require different clean up steps.

1.2.2. Inclination Detection

The second step is to determine the possible inclinations of the four line segments of a possible anticrossing. We perform a (deterministic) Hough line transform for angles between $-110°$ and $30°$, as this range covers the inclinations of typical anticrossings. The hough_line_peaks method from

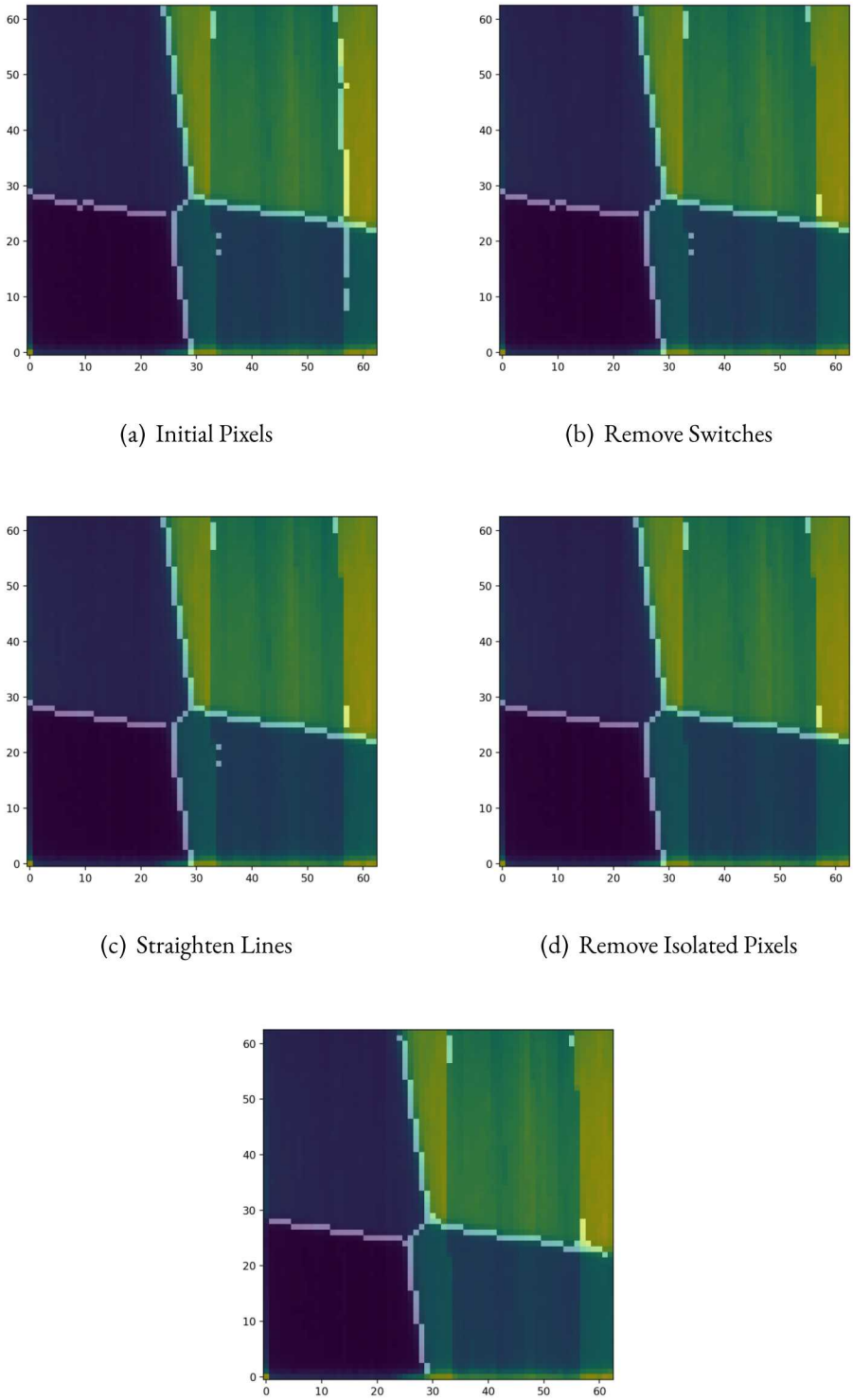
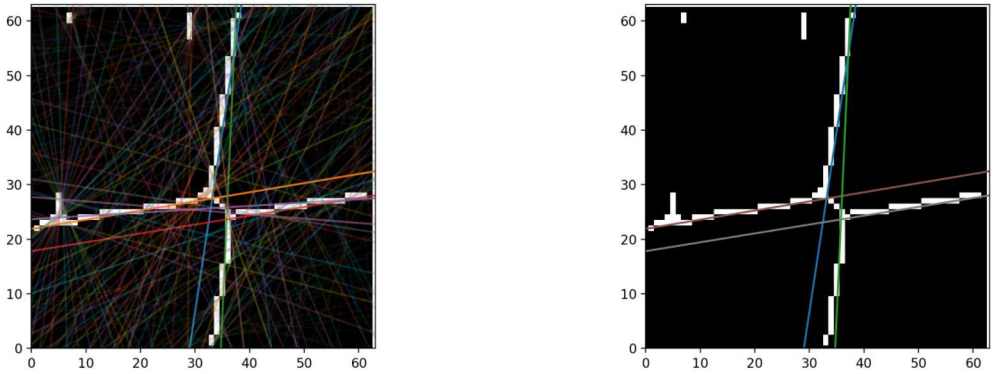


Figure 1-3. After forming the initial binary image (a), we remove switches (b), straighten lines (c), remove isolated pixels (d), and perform binary closing (e) to get the cleaned binary image.

Scikit-image returns a list of lines in the image which we score based on how well a template of the line overlaps with the binary image.

We then filter through the detected lines in order of decreasing score, and retain lines with a minimum distance to the image center of less than 25 percent of the image size. When we select a line, we loop through the remaining lines and discard any lines where 70% of the pixels in the overlap between the line and the binary image are also in the overlap of the previously selected lines and the binary image. See figure 1-4 for an example. We stop when we have considered all detected lines, or when we have selected at least two lines and the score of the current line is less than half the score of the highest scoring line.



(a) All Lines Returned

(b) Selected Lines

Figure 1-4. On the left, we have all lines returned by the Hough transform where the transparency is determined by the score. On the right, we have only the lines retained after filtering. Note the data has been flipped in order to conform with the origin location in Scipy's implementation of the Hough line transform.

Finally, we convert each selected line to an inclination and split these into two groups, Φ and Θ with $\min \Phi > \max \Theta$, using k-means with $k = 2$. Performing a clustering algorithm, as opposed to calculating a threshold, allows for arbitrarily subtle differences between the two groups of angles.

1.2.3. Hough Anticrossing Transform

Third, we generalize the Hough transform to detect the locations and inclinations of possible triple points. Let $\mathcal{P}$ be the set of all white pixels and $\mathcal{C}$ be the set of white pixels in the middle M percent of the image. We initialize four two-dimensional accumulators, $A_{\theta_u}, A_{\phi_u}, A_{\theta_d}, A_{\phi_d}$, where the rows correspond to the elements of $\mathcal{C}$ and the columns correspond to the elements of either Θ or Φ. We proceed by algorithm 1 which determines an association between white pixels and anticrossing parameters.

Algorithm 1 Hough-like Anticrossing Transform

for all $(p, c) \in \mathcal{P} \times \mathcal{C}$ **do**
  if distance$(p, c) <$ anticrossing_length **then**
    $\omega \leftarrow$ inclination of the line between p and c
    if $\omega \in \Phi$ **then**
      if p is right of c **then**
        $A_{\phi_u}[c, \omega] \leftarrow A_{\phi_u}[c, \omega] + 1$
      else
        $A_{\phi_d}[c, \omega] \leftarrow A_{\phi_d}[c, \omega] + 1$
      end if
    else if $\omega \in \Theta$ **then**
      if p is right of or directly below c **then**
        $A_{\theta_d}[c, \omega] \leftarrow A_{\theta_d}[c, \omega] + 1$
      else
        $A_{\theta_u}[c, \omega] \leftarrow A_{\theta_u}[c, \omega] + 1$
      end if
    end if
  end if
end for

Finally, we create a three-dimensional accumulator for each triple point whose dimensions correspond to entries of $\mathcal{C}$, Θ, and Φ with

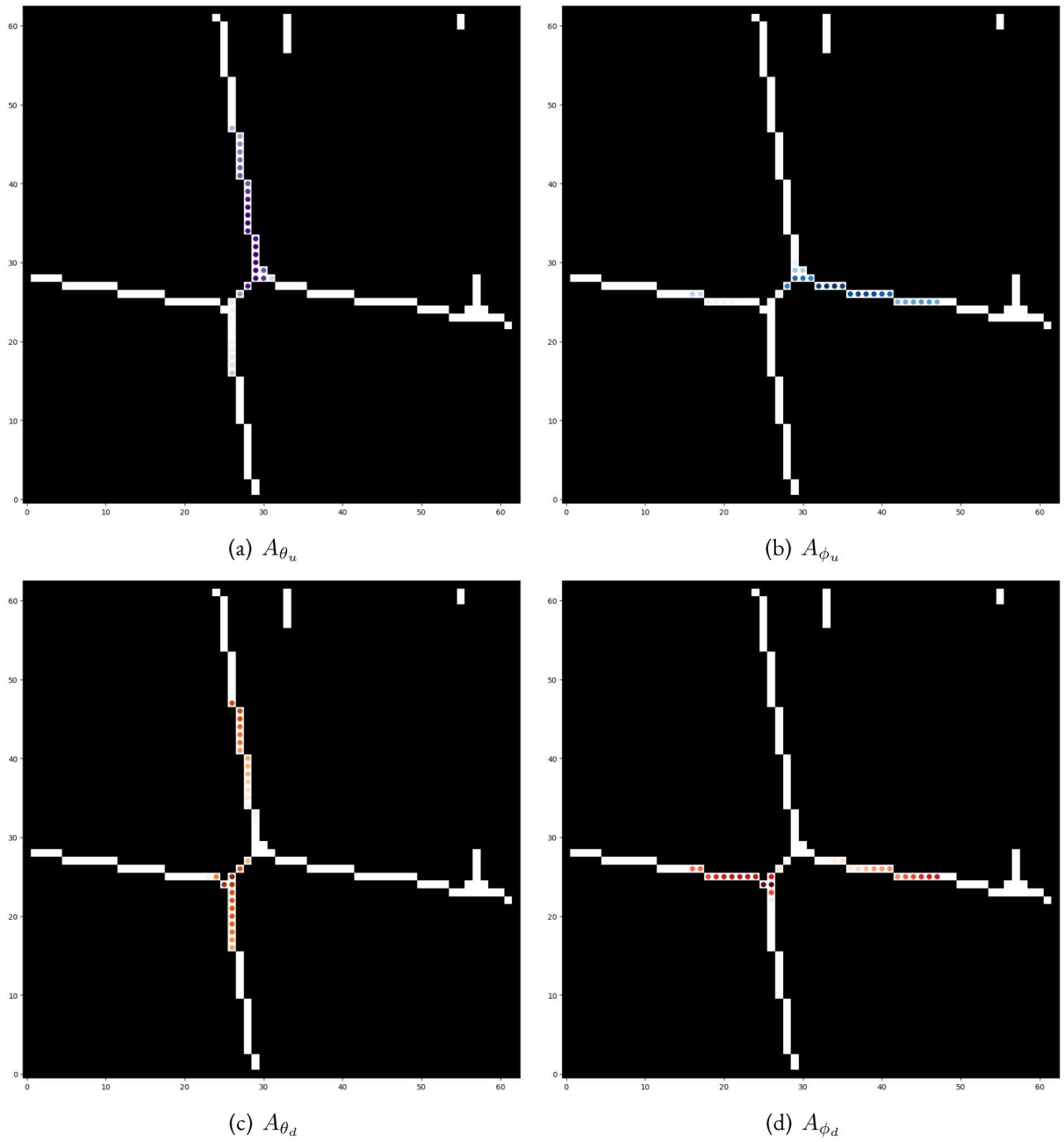
$$A_u[c, \theta, \phi] = A_{\theta_u}[c, \theta] + A_{\phi_u}[c, \phi]$$
$$A_d[c, \theta, \phi] = A_{\theta_d}[c, \theta] + A_{\phi_d}[c, \phi].$$

Figure 1-5 depicts the results of algorithm 1, and figure 1-6 depicts the results after creating the three-dimensional accumulators.

1.2.4. Select Triple Points

Fourth, we extract triple points from the final accumulators and pair them into anticrossings. We start by creating a list of the locations of local maximums of A_u and A_d as possible candidates for each triple point. We then create a list of potential candidates for the anticrossing $A = (p_u, \theta_u, \phi_u, p_d, \theta_d, \phi_d)$ by checking the assumptions listed above. We score each valid candidate by how well a template created from the parameters overlaps with the binary image. The template created contains the line connecting the two triple points. If this line is visible in the binary image, it will reward candidate anticrossings that line up with it, without punishing scans where the line is not visible.

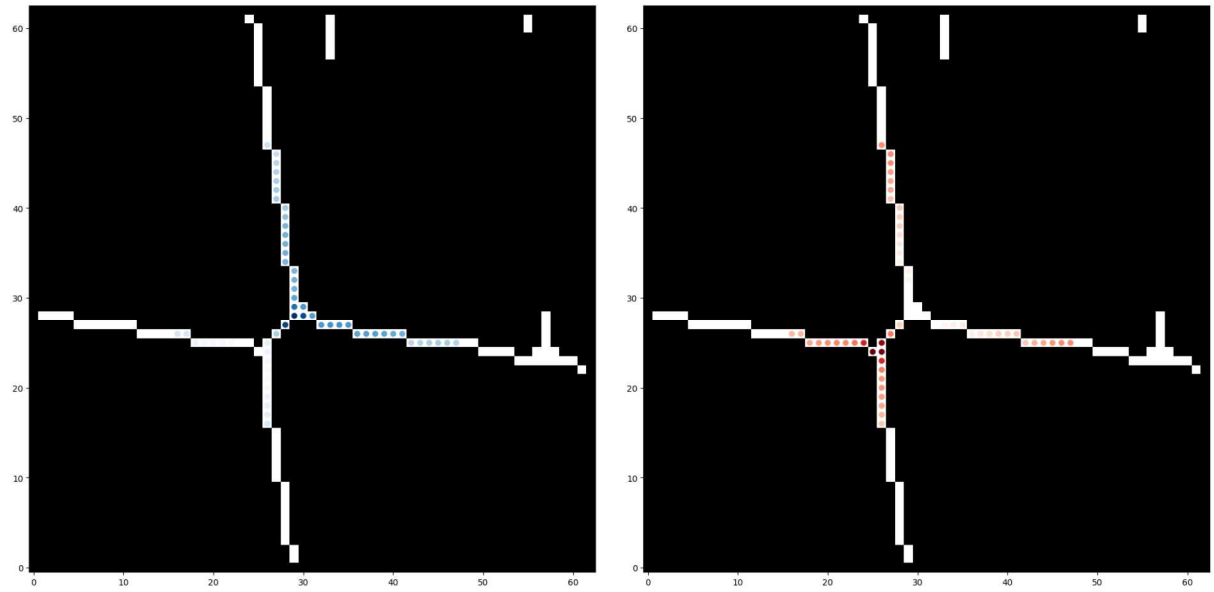
When we are only searching for a single anticrossing, we select the anticrossing with the highest score. When we are searching for multiple anticrossings, we proceed through the candidates in order of decreasing score. Whenever we select an anticrossing, we discard candidates that have a triple point near one of the triple points of the selected anticrossings. Figure 1-7 depicts the anticrossing selected from the accumulators in figure 1-6.



(a) A_{θ_u} (b) A_{ϕ_u}

(c) A_{θ_d} (d) A_{ϕ_d}

Figure 1-5. The four accumulators after the Hough transform. Each pixel p is colored based on the maximum score of (p, ω) for some ω in the indicated accumulator.



(a) A_u (b) A_d

Figure 1-6. The results after the four accumulators have been consolidated into one for each triple point. Each pixel p is colored based on the maximum score of (p, θ, ϕ) for some θ and ϕ in the indicated accumulator.

1.2.5. Optimization

The previous steps give a list of anticrossings in the image, but there are at least two sources of imprecision that lead to sub-optimal results. The first is that when dealing with integer pixel locations, the inclination of a line is only approximately the inclination between any two pixels on the line. The second is that the above algorithm can only select triple point locations from the set of white pixels. Thus the fifth step is to perform a local search to optimize the parameters of the anticrossing(s).

For a pixel location p and an inclination ω, let $s(p, \omega)$ be the overlap between the binary image and a template created of a line segment with those parameters. Then let

$$s(p) = \max_{\theta} s(p, \theta) + \max_{\phi} s(p, \phi).$$

Separately for each triple point P in a detected anticrossing, we search for the pixel location p with maximum $s(p)$, as well as the θ and ϕ that realize the maximum, as follows. Let Q be a priority queue that holds pixel locations to search. Initialize Q with the location of P. Until the search stops, pop off the first element of Q, denoted q and calculate $s(q)$. Then add the neighbors of q to Q with priority $s(q)$. The search stops when Q is empty or a predetermined number of pixels have been searched. At this point, the highest-scoring positions and inclinations are returned. Figure 1-8 depicts the results of optimizing the anticrossing in figure 1-7.

Two passes are done for the optimization – one where the inclinations of the two triple points are forced to be the same and one where the inclinations are allowed to differ slightly.

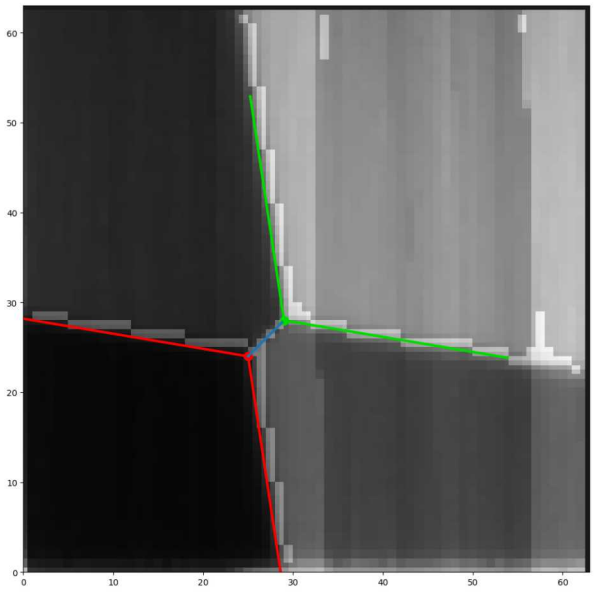


Figure 1-7. The selected anticrossing formed from the accumulators in figure 1-6.

1.3. RESULTS AND CHALLENGES

After developing our algorithm on approximately 50 scans, we then received additional data giving us about 1400 scans that contain a single, mostly central anticrossing. The algorithm did not change much after we received the additional data – some parameters were adjusted and operational improvements to the code were made. Of the 1400 scans, approximately 1300 returned good results (where good is relative to the original scan quality). Of the remaining 100, about half returned no results and half returned an incorrect anticrossing. Additionally, for approximately 85% of the scans, the algorithm completes in less than 5 seconds and for approximately 99% of the scans, in less than 30. Rare cases can take up to 15 minutes, though usually this is a sign that something has gone wrong.

The main factors that influence the performance of the algorithm are

- the resolution, in terms of number of pixels, of the scan,
- the amount of noise in the image that affects the visibility of the anticrossing,
- the algorithm parameters being appropriately set, and
- the clarity of the shifts between regions that correspond to electron transitions.

The algorithm will not necessarily fail if these conditions are not true, but it is more likely to either fail or return degraded results. Figure 1-9 has examples where the algorithm performs well, including a few cases where these conditions don't all hold.

On the other hand, figure 1-10 gives examples where something in the algorithm has gone wrong that exemplify the conditions listed above:

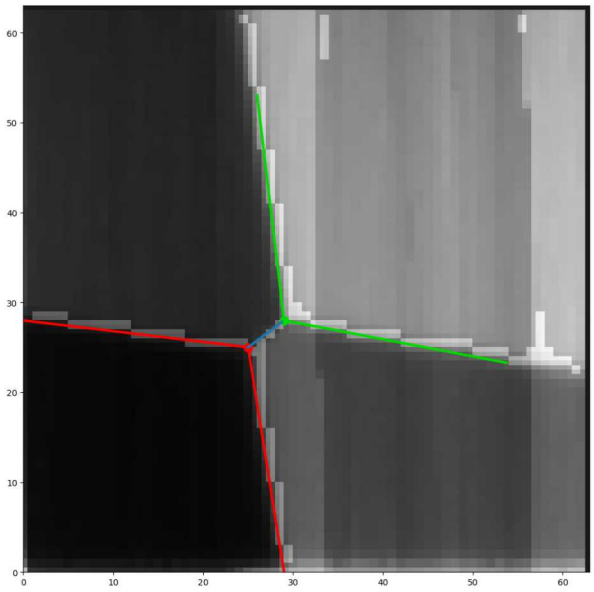


Figure 1-8. The anticrossing from figure 1-7 after optimization.

- Scan 1-10(a) has failed because the low resolution of the scan and the fuzzy vertical transitions have caused the binary to miss the vertical legs.
- Scan 1-10(b) has failed because noise has introduced false lines in the binary image.
- Scan 1-10(c) has failed because the upper triple point is too close to the edge of the image and is being excluded by the parameters.
- Scan 1-10(d) has failed because fuzzy horizontal transitions have caused the binary to miss the horizontal legs.
- Finally, scan 1-10(e) has failed because noise in the scan has caused the binary to miss a leg and the low resolution has distorted the detected angles.

Figures 1-9 and 1-10 contain the data overlaid with the binary image and the results returned by the algorithm to better illustrate the algorithm's performance. The formation of the binary image is the most critical and delicate step of the algorithm – it is most likely to fail and most likely to determine the quality of the final results.

1.3.1. Multiple Anticrossings

While the bulk of our development and testing has been on the case of a single anticrossing, the same algorithm works for detecting multiple anticrossings in a single scan. Figure 1-11 contains two examples of where the algorithm works well on multiple anticrossings, while figure 1-12 shows two where the algorithm struggles.

The algorithm struggles with multiple anticrossings for the following reasons. First, parameters are set automatically as a percentage of the image size assuming a single central anticrossing. These parameters

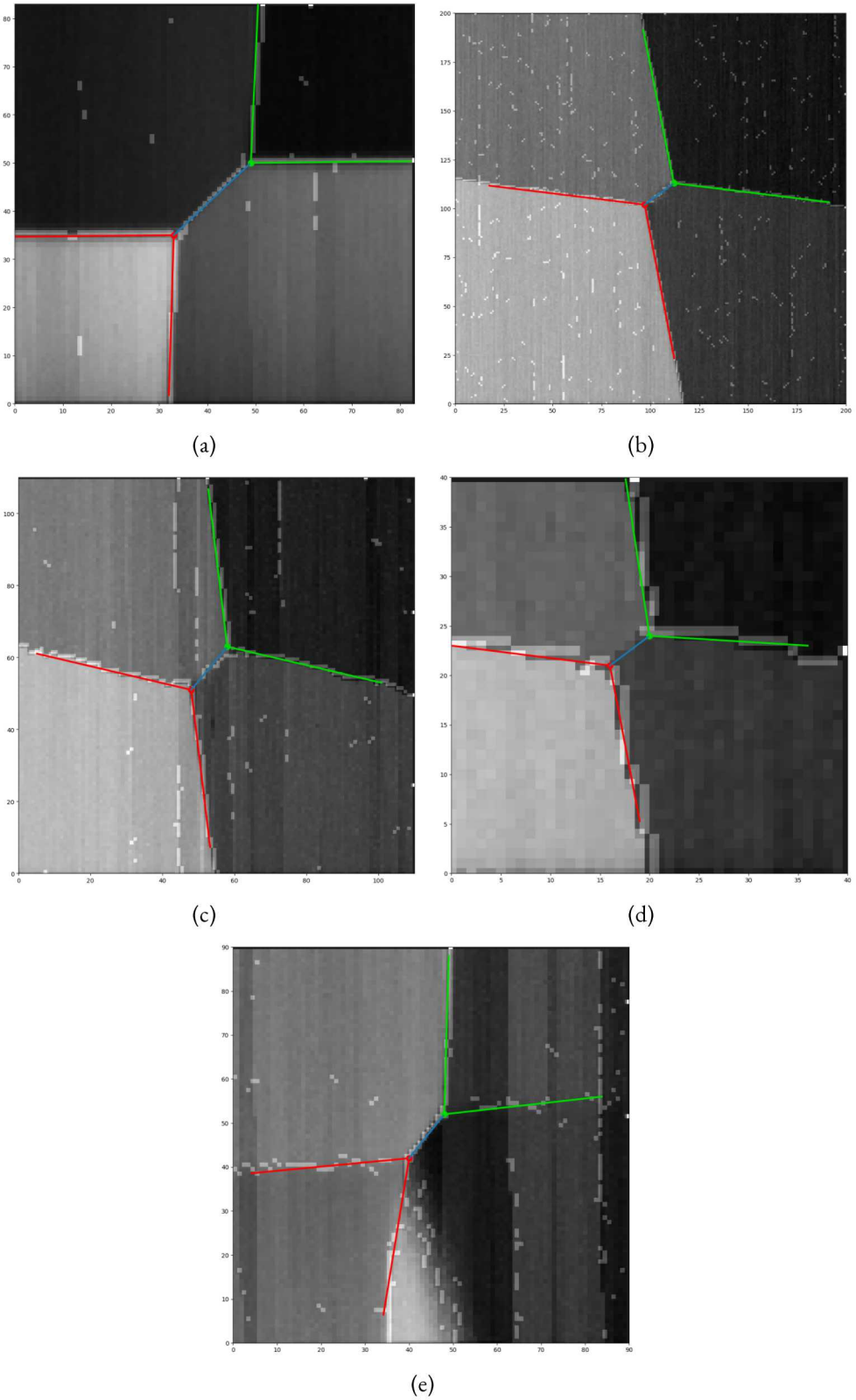


Figure 1-9. In all of these examples, the algorithm successfully identifies the anticrossing and its parameters. Scan (d) is an example of successful performance despite low resolution while scan (e) contains significant noise interfering with the visibility of the anticrossing.

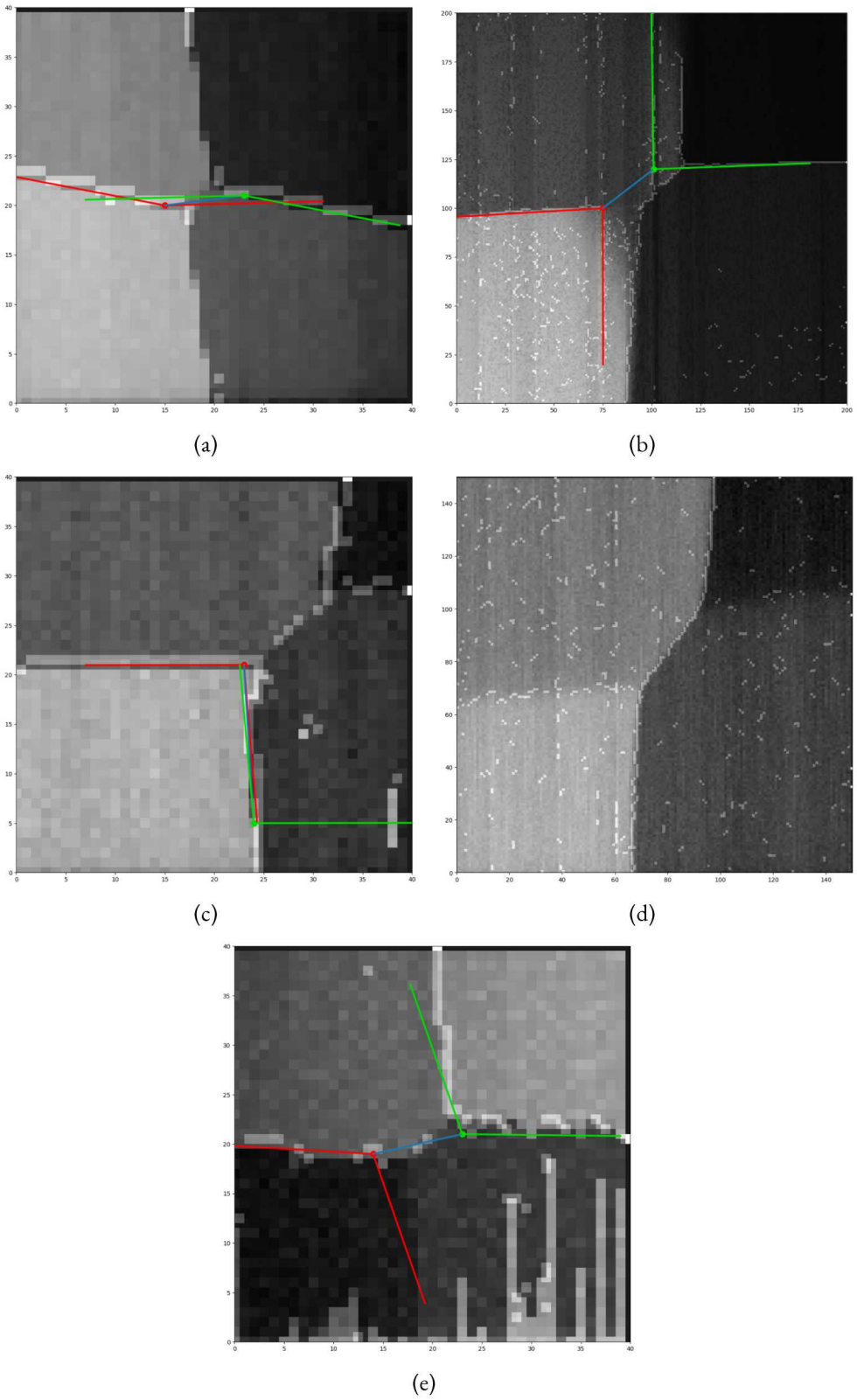
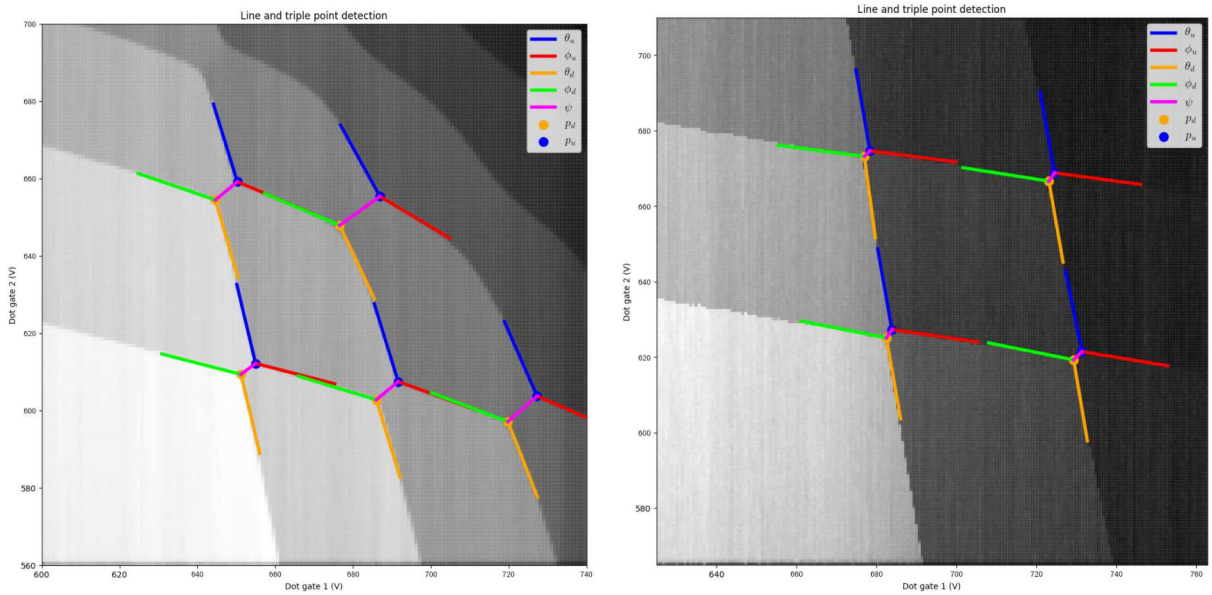
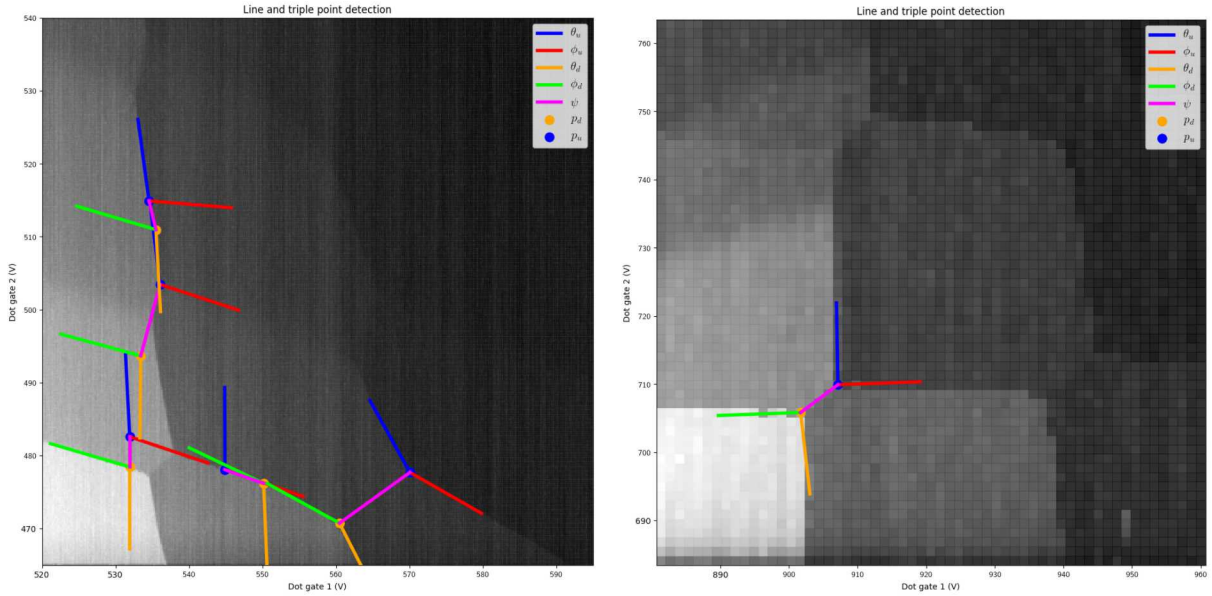


Figure 1-10. In all of these examples, the algorithm fails to identify the anticrossing or at least one of the parameters.



(a) (b)

Figure 1-11. Here the algorithm is able to successfully identify multiple anticrossings in the scans.



(a) (b)

Figure 1-12. Here the algorithm either fails to identify anticrossings or returns false positives.

can be adjusted for multiple anticrossings (and need to be), but what they should be adjusted to depends on the number of anticrossings in the image. Thus, in order for this mode to be more robust, either the user would need to specify the number of anticrossings (which is not useful for an automated system), or this number would need to be detected automatically.

Second, the pairing algorithm described in section 1.2.4 was designed with a single anticrossing in mind. Since false positives will not match as strongly as the actual anticrossing, returning the anticrossing with the highest score naturally removes false positives. When multiple anticrossings are being detected, more care needs to be taken in pairing triple points together and discarding false triple points.

Finally, the formation of the binary image described in section 1.2.1 was designed under the assumption that each row and column would have a few white pixels. This should be true when the scan contains a single anticrossing, but with more anticrossings there are more interesting pixels.

Furthermore, as it is currently implemented the algorithm can be very slow in scans with multiple anticrossings. There are a couple of points that are prime targets for parallelism but these have not been implemented or tested.

2. IMAGING WITH NITROGEN-VACANCY DIAMONDS

2.1. INTRODUCTION

Diamonds consist of a lattice of carbon atoms. A nitrogen-vacancy center is a naturally occurring flaw in diamonds where one of the carbon atoms is replaced by a nitrogen atom next to a vacancy, a spot in the lattice that is missing its carbon atom. Nitrogen-vacancy centers fluoresce when illuminated and are also paramagnetic, meaning that they temporarily respond weakly to magnetic fields in the direction of the applied magnetic field. These two facts combine to mean that we can use them to sense magnetic fields. In fact, nitrogen-vacancy centers can be placed very close to the diamond surface, and thus the magnetic sample, which gives better resolution than other methods for imaging magnetic fields [6].

To image the magnetic field, a microwave probe is applied at a sweep of frequencies and the fluorescence intensity is measured at each frequency for each pixel. Figure 2-1 depicts a typical spectrum obtained with this method. The two sections denoted in red are of interest. They can each be modeled by a triple

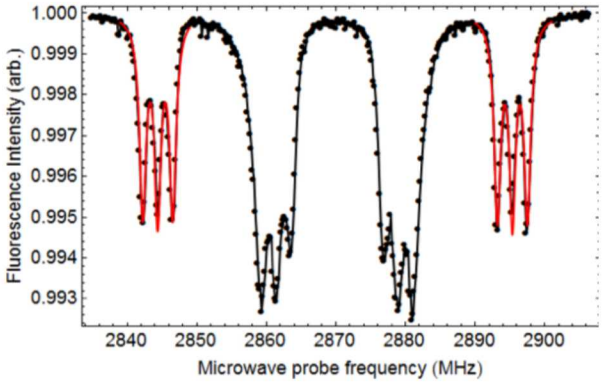


Figure 2-1. The spectrum for a single pixel of imaging a magnetic field using nitrogen vacancy centers. Image by P. Kehayias.

Lorentzian function

$$y = 1 + y_0 - \frac{a_1 w^2}{(x-(x_0-h))^2+w^2} - \frac{a_2 w^2}{(x-x_0)^2+w^2} - \frac{a_3 w^2}{(x-(x_0+h))^2+w^2} \tag{2.1}$$

where h is known from the physics, but $x_0, y_0, w, a_1, a_2, a_3$ are all unknown. The magnetic field at the pixel is proportional to the difference in the central frequency x_0 of each of the red regions.

Traditionally, the magnetic field is found by fitting equation 2.1 to each of the red regions, subtracting the resulting central frequencies, and scaling by the appropriate factor. While grounded in the physics of the system, this process is slow and takes up to an hour to produce the image after the scan. Because the imaging setup is delicate, often bad data is collected, and the experimenter does not know this until an hour has passed, slowing up research progress.

The goal of this chapter is to present a quick approach that produces an approximate image. This allows the experimenter to quickly check the image, and make appropriate adjustments to the setup. When the image is as desired, the slower, theoretically-based method of Lorentzian fits can be run for higher precision in the results.

2.2. APPROACH

One reason that the Lorentzian fit is slow is that six parameters are fit from equation 2.1 for each side, but only the central frequency is actually necessary for determining the image. Furthermore, we only care about the difference in the central frequencies as opposed to the actual values. We propose using cross-correlation to find the optimal shift between the two spectra.

Cross-correlation measures the similarity of two signals as a function of the shift of one relative to the other. For discrete, real-valued signals f and g, the cross-correlation is defined as the function of the shift n

$$(f \star g)[n] = \sum_{m=-\infty}^{\infty} f(m)g(m+n),$$

i.e. $f \star g$ is the "sliding dot product" of f against g. Thus, the n that yields the maximum of $f \star g$ is the shift that best lines up the peaks of each. Cross-correlation can be calculated by the Fourier transform, for which there exist very fast algorithms.

The outline of our approach is as follows. For each pixel p, let $f(p)$ be the spectrum on the left and $g(p)$ be the spectrum on the right. Then let

$$f'(p) = 1 - f(p) \qquad\qquad g'(p) = 1 - g(p).$$

This inverts the spectra, meaning we are lining up peaks instead of valleys. Now let $F(p)$ and $G(p)$ be formed by linearly interpolating the points of f' and g' to allow for greater precision. Finally, let

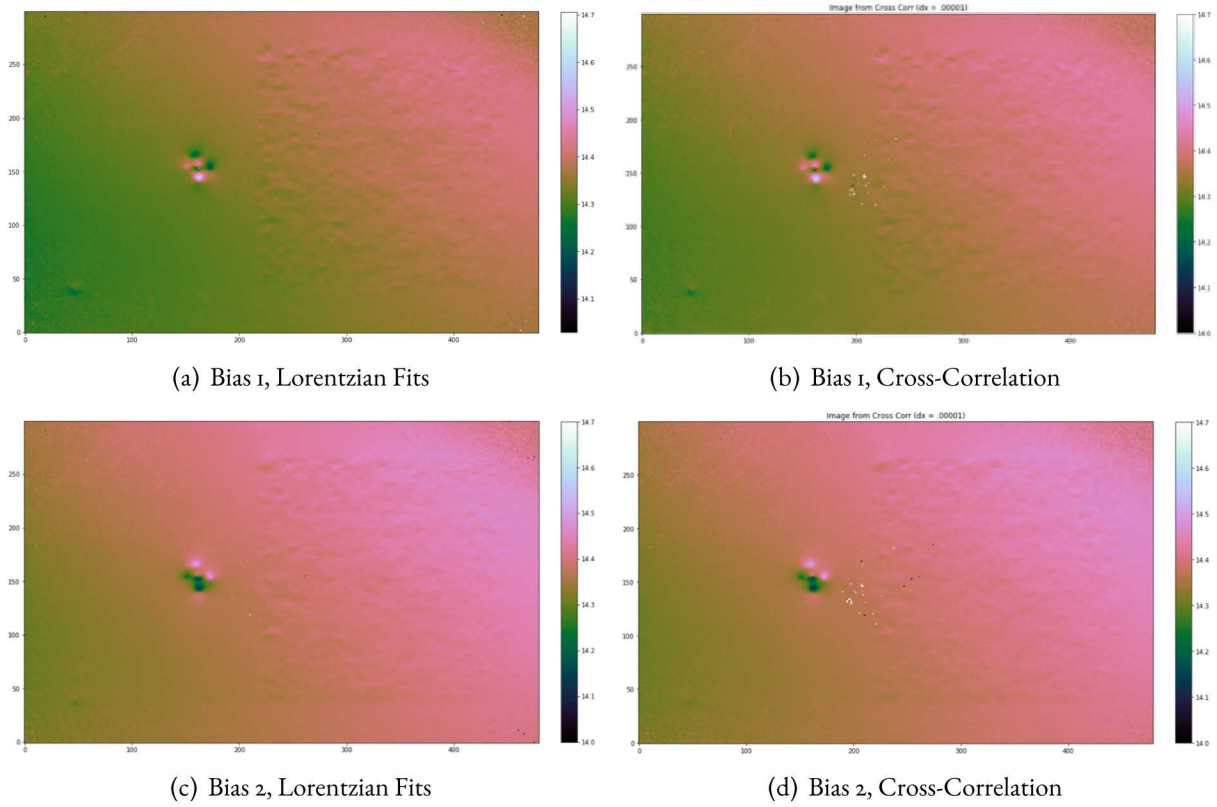
$$x(p) = \max(F(p) \star G(p)),$$

and let $m(p)$ be the value of the magnetic field formed from appropriately scaling $x(p)$.

We form the image by first binning the spectra for pixels in a 4×4 square in order to increase the signal-to-noise ratio and cut down on the number of pixels to be calculated. Then for each of the binned pixels, we calculate $m(p)$ as above.

Without the step of linear interpolation forming $F(p)$ and $G(p)$ we were unable to get enough precision on the correlation placement to produce an image with high fidelity to the original image. Figure 2-2 has a comparison of the images formed by the Lorentzian fits and the cross-correlation

method described above with linear interpolation to a step size of $dx = 0.00001$ on two different scans of the same magnetic field with different bias fields applied. Figure 2-3 has the the difference image between the Lorentzian fits and the cross-correlation. The mean squared error corresponding to this figure is 2.3039×10^{-5}.



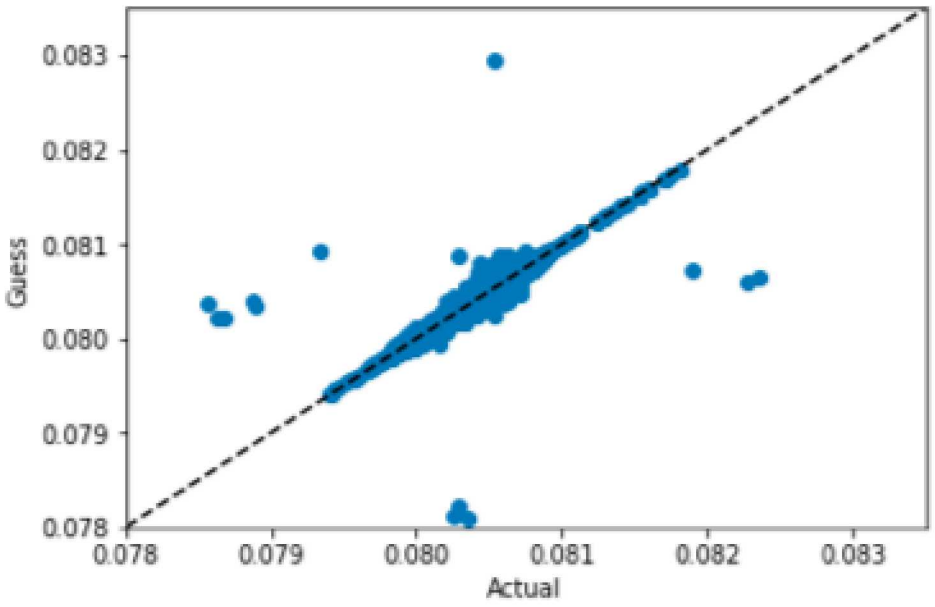
(a) Bias 1, Lorentzian Fits

(b) Bias 1, Cross-Correlation

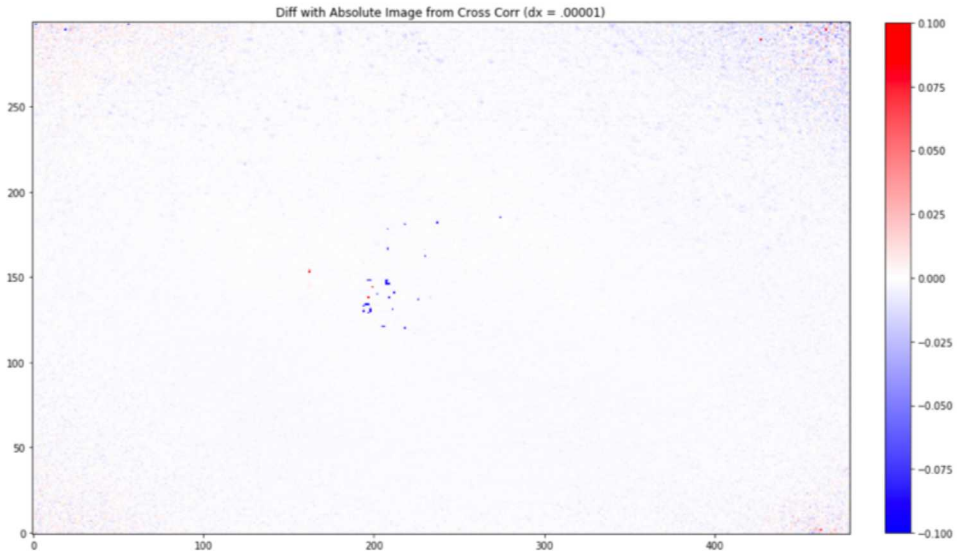
(c) Bias 2, Lorentzian Fits

(d) Bias 2, Cross-Correlation

Figure 2-2. Images formed from two different scans of the same field with different bias fields applied.



(a) Cross-Correlation vs Lorentzian Fits



(b) Difference Image

Figure 2-3. The error in using cross-correlation to approximate Lorentzian fits.

REFERENCES

[1] D.H. Ballard. Generalizing the hough transform to detect arbitrary shapes. *Pattern Recognition*, 13(2):111 – 122, 1981.

[2] J. E. Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems Journal*, 4(1):25–30, 1965.

[3] R. Duda and P. Hart. Use of the Hough transformation to detect lines and curves in pictures. *Commun. ACM*, 15(1):11–15, January 1972.

[4] R. P. Feynman. Simulating physics with computers. *International Journal of Theoretical Physics*, 21(6):467–488, Jun 1982.

[5] P. Hough. Method and means for recognizing complex patterns, U.S. Patent 3,069,654, Dec. 18, 1962.

[6] K. Jensen, N. Leefer, A. Jarmola, Y. Dumeige, V. M. Acosta, P. Kehayias, B. Patton, and D. Budker. Cavity-enhanced room-temperature magnetometry using absorption by nitrogen-vacancy centers in diamond. *Phys. Rev. Lett.*, 112:160802, Apr 2014.

[7] N. Kiryati, Y. Eldar, and A.M. Bruckstein. A probabilistic Hough transform. *Pattern Recognition*, 24(4):303 – 316, 1991.

[8] D. Loss and D. P. DiVincenzo. Quantum computation with quantum dots. *Phys. Rev. A*, 57:120–126, Jan 1998.

[9] AR Mills, MM Feldman, C Monical, PJ Lewis, KW Larson, AM Mounce, and JR Petta. Computer-automated tuning procedures for semiconductor quantum dot arrays. *Applied Physics Letters*, 115(11):113501, 2019.

[10] J. R. Petta, A. C. Johnson, J. M. Taylor, E. A. Laird, A. Yacoby, M. D. Lukin, C. M. Marcus, M. P. Hanson, and A. C. Gossard. Coherent manipulation of coupled electron spins in semiconductor quantum dots. *Science*, 309(5744):2180–2184, 2005.

[11] P. W. Shor. Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. *SIAM J. Comput.*, 26(5):1484–1509, October 1997.

APPENDICES

A. CODE SNIPPETS OF ANTICROSSING DETECTION

This appendix contains code snippets that should allow the reader to implement the algorithm themselves. For a more high-level description of the algorithm see section 1.2 and for a complete description of the different algorithm parameters see appendix B.

NumPy is generally imported as "np." The only code included is code that is implementing the algorithms described – code specific to the data format or the interface as been omitted as it is use case specific. The main function takes a numpy array containing the data named "data." Unless otherwise specified, the SciPy or NumPy implementations of standard algorithms are used.

A. PARAMETERS

This section contains the parameter objects that control the operation of the algorithms

```
class AShapeParameters:
    def __init__(self,anticrossing_length=.4,flat_buffer=10,between_buffer=30):
        self.anticrossing_length = anticrossing_length
        self.flat_buffer = flat_buffer
        self.between_buffer = between_buffer

class BinaryParameters:
    def __init__(self,grad_percentile=95,border=5,peak_width=5,switch_perc=.6,
                 white_col_perc=.1,isolated_thresh=2,remove_switches=True,
                 binary_closing_iterations=0,straighten_lines=True):
        self.grad_percentile = grad_percentile
        self.border = border
        self.peak_width = peak_width
        self.switch_perc = switch_perc
        self.white_col_perc = white_col_perc
        self.isolated_thresh = isolated_thresh
        self.remove_switches = remove_switches
        self.binary_closing_iterations = binary_closing_iterations
        self.straighten_lines = straighten_lines

class OptimizationParameters:
```

```
22      def __init__(self,max_angle_change=45,num_angle_steps=60,max_point_dist=10,
23                   num_to_search=50,thickness=1):
24          self.max_angle_change = max_angle_change
25          self.num_angle_steps = num_angle_steps
26          self.max_point_dist = max_point_dist
27          self.num_to_search = num_to_search
28          self.thickness = thickness
29
30      def small_optimization():
31          return OptimizationParameters(5,50,3,10)
32
33  class HoughParameters:
34      def __init__(self,min_dist_to_vote=4,match_angle_tolerance=5.0,middle=.5):
35          self.min_dist_to_vote = min_dist_to_vote
36          self.match_angle_tolerance = match_angle_tolerance
37          self.middle = middle
38
39  class APairParameters:
40      def __init__(self,neighborhood=5,anticrossing_width_max=.5,
41                   same_angle_tolerance=25,quality=.5,anticrossing_width_min=2):
42          self.neighborhood = neighborhood
43          self.anticrossing_width_max = anticrossing_width_max
44          self.anticrossing_width_min = anticrossing_width_min
45          self.same_angle_tolerance = same_angle_tolerance
46          self.quality = quality
47
48  class TriplePoint:
49      def __init__(self,point,lower,angles):
50          self.lower = lower
51          self.point = point
52          self.horiz = angles[0]
53          self.vert = angles[1]
54
55      def left(self):
56          if self.lower: return self.horiz+180
57          else: return self.vert+180
58
59      def right(self):
60          if self.lower: return self.vert
61          else: return self.horiz
62
63  class Anticrossing:
64      def __init__(self,lower_point,lower_angles,upper_point,upper_angles):
65          self.lower = TriplePoint(lower_point,True,lower_angles)
66          self.upper = TriplePoint(upper_point,False,upper_angles)
```

```
67
68  class Angles:
69      def __init__(self,horiz_angles,vert_angles):
70          self.horiz_angles = sorted(horiz_angles)
71          self.vert_angles = sorted(vert_angles)
72          if len(self.horiz_angles) > 0 and len(self.vert_angles) > 0:
73              temp = [np.max(self.vert_angles),np.min(self.horiz_angles)]
74              self.split_angle = np.mean(temp)
75          else: self.split_angle = -45
```

B. TEMPLATE CREATION

Template creation uses the Bresenham algorithm for drawing pixelated lines [2]. Thickening the lines drawn is done with SciPy's Gaussian filter. When profiling the code, a significant portion of the code is spent applying the Gaussian filter.

```
1   class bresenham:
2       def __init__(self, start, end):
3           self.start = list(start)
4           self.end = list(end)
5           self.path = []
6
7           dy = self.end[1]-self.start[1]
8           dx = self.end[0]-self.start[0]
9           self.steep =  abs(dy) > abs(dx)
10
11          if self.steep:
12              self.start = self.swap(self.start[0],self.start[1])
13              self.end = self.swap(self.end[0],self.end[1])
14
15          if self.start[0] > self.end[0]:
16              _x0 = int(self.start[0])
17              _x1 = int(self.end[0])
18              self.start[0] = _x1
19              self.end[0] = _x0
20
21              _y0 = int(self.start[1])
22              _y1 = int(self.end[1])
23              self.start[1] = _y1
24              self.end[1] = _y0
25
26          dx = self.end[0] - self.start[0]
27          dy = abs(self.end[1] - self.start[1])
28
```

```
        with np.errstate(divide='ignore',invalid='ignore'):
            error = 0
            derr = dy/float(dx)

            ystep = 0
            y = self.start[1]

            if self.start[1] < self.end[1]: ystep = 1
            else: ystep = -1

            for x in range(self.start[0],self.end[0]+1):
                if self.steep:
                    self.path.append((y,x))
                else:
                    self.path.append((x,y))

                error += derr
                if error >= 0.5:
                    y += ystep
                    error -= 1.0

    def swap(self,n1,n2):
        return [n2,n1]

def find_endpoint(angle, length):
    endy = length * math.cos(math.radians(angle))
    endx = length * math.sin(math.radians(angle))
    return endx, endy

#flips x and y because will take transpose
def find_endpoint_from_start(point,angle,length):
    endy, endx = find_endpoint(angle,length)
    return (point[0]+endx,point[1]+endy)

def create_template_line_two_points(xsize,ysize,point1,point2,
                                    thickness,binary=True):
    xsize, ysize = ysize, xsize
    point1 = (int(round(point1[0])),int(round(point1[1])))
    point2 = (int(round(point2[0])),int(round(point2[1])))
    line = bresenham(point1,point2).path
    template = np.zeros([xsize,ysize],dtype=int)
    for x in line:
        if (x[0] >= 0 and x[0] < xsize and x[1] >= 0 and x[1] < ysize):
            template[x] = 255
    if thickness > 0: template = gaussian_filter(template, sigma=thickness)
```

```
    if binary: template[template > 0] = 1
    template = template.transpose()
    return template

def create_template_line_full_image(xsize,ysize,point,angle,
                                    length,thickness,binary=True):
    end = find_endpoint_from_start(point,angle,length)
    return create_template_line_two_points(xsize,ysize,
        point,end,thickness,binary)

def create_template_anticrossing(xsize,ysize,anticrossing,
                                 length,thickness,binary=True):
    template = np.zeros([xsize,ysize], dtype=int)

    template += create_template_line_full_image(xsize,ysize,
        anticrossing.lower.point,anticrossing.lower.left(),length,thickness,False)
    template += create_template_line_full_image(xsize,ysize,
        anticrossing.lower.point,anticrossing.lower.right(),length,thickness,False)
    template += create_template_line_full_image(xsize,ysize,
        anticrossing.upper.point,anticrossing.upper.left(),length,thickness,False)
    template += create_template_line_full_image(xsize,ysize,
        anticrossing.upper.point,anticrossing.upper.right(),length,thickness,False)
    template += create_template_line_two_points(xsize,ysize,
        anticrossing.lower.point,anticrossing.upper.point,thickness,False)

    if binary: template[template > 0] = 1
    return template
```

C. BINARY FORMATION

This section contains the code for the binary formation algorithm in section 1.2.1.

```
def normalize(arr):
    min_v = np.min(arr)
    max_v = np.max(arr)

    ans = arr - min_v
    ans /= (max_v-min_v)

    return ans

def make_binary(data,params):
    points = {}
```

```
    rows = []
    cols = []

    normalized = normalize(da.data)
    n,m = normalized.shape
    for i in range(n): rows.append(abs(np.gradient(normalized[i,:])))
    for i in range(m): cols.append(abs(np.gradient(normalized[:,i])))

    for i in range(n):
        row = rows[i]
        thresh = np.percentile(row,params.grad_percentile)
        row[row < thresh] = 0
        for j in range(params.border,m-params.border):
            if row[j] > 0 and
               row[j] == max(row[j-params.peak_width:j+params.peak_width+1]):
                if i not in points: points[i] = []
                if j not in points[i]: points[i].append(j)

    for i in range(m):
        col = cols[i]
        thresh = np.percentile(col,params.grad_percentile)
        col[col < thresh] = 0
        for j in range(params.border,n-params.border):
            if col[j] > 0 and
                col[j] == max(col[j-params.peak_width:j+params.peak_width+1]):
                if j not in points: points[j] = []
                if i not in points[j]: points[j].append(i)

    #remove long (mostly) horizontal lines
    if params.remove_switches:
        to_remove = []
        for row in points:
            cols = []
            for col in points[row]:
                cols.append(col)
            if row+1 in points:
                for col in points[row+1]:
                    if col not in cols: cols.append(col)
            if len(cols) > m*params.switch_perc:
                if row in points and row not in to_remove: to_remove.append(row)
                if row+1 in points and row+1 not in to_remove:
                    to_remove.append(row+1)
        for row in to_remove:
            to_remove_col = []
            for col in points[row]:
```

```
                count = 0
                for row2 in points:
                    if col in points[row2]: count += 1.0
                if count/n < params.white_col_perc: to_remove_col.append(col)
            for col in to_remove_col: points[row].remove(col)

    #straighten lines
    if params.straighten_lines:
        for row in points:
            cols = points[row][:]
            for col in cols:
                c2 = -1
                c3 = -1
                if row-1 in points:
                    for col2 in points[row-1]:
                        if abs(col2-col) <= 1:
                            c2 = col2
                if row+1 in points:
                    for col3 in points[row+1]:
                        if abs(col3-col) <= 1:
                            c3 = col3
                if c2 == c3 and c2 != -1 and c2 != col and c2 not in points[row]:
                    points[row].remove(col)
                    points[row].append(c2)
                    continue

                if row-1 in points:
                    if col-1 in points[row-1] and col+1 in points[row-1] and
                       col not in points[row-1]:
                        points[row].remove(col)
                        points[row-1].append(col)
                        continue

                if row+1 in points:
                    if col-1 in points[row+1] and col+1 in points[row+1] and
                        col not in points[row+1]:
                        points[row].remove(col)
                        points[row+1].append(col)
                        continue

    #remove isolated points
    for i in points:
        if i < params.border or i > n-params.border: continue
        to_remove = []
        for j in points[i]:
```

```
            if j < params.border or j > m-params.border: continue
            count = 0
            for row in range(i-1,i+2):
                if row not in points: continue
                for col in range(j-1,j+2):
                    if col in points[row]: count += 1
            if count <= params.isolated_thresh: to_remove.append(j)
        for col in to_remove: points[i].remove(col)

    binary = np.zeros((n,m))
    for i in points:
        for j in points[i]:
            binary[i,j] = 1

    if params.binary_closing_iterations > 0:
        binary = binary_closing(binary,iterations=params.binary_closing_iterations)
        binary = binary.astype(int)

    return binary
```

D. INCLINATION DETECTION

This section contains the code for the inclination detection algorithm described in section 1.2.2.

```
def group_angles(angles):
    if len(angles) < 2: return Angles([],[])
    kmeans = KMeans(2).fit(np.array(angles).reshape(-1,1))
    angle_groups = []
    for i in range(len(angles)):
        label = kmeans.labels_[i]
        while label >= len(angle_groups): angle_groups.append([])
        angle = angles[i]
        angle_groups[label].append(angle)

    angle_means = [(np.mean(angle_groups[i]),i) for i in range(len(angle_groups))]
    angle_means.sort()

    vert_angles = angle_groups[angle_means[0][1]]
    if len(angle_means) > 1: horiz_angles = angle_groups[angle_means[1][1]]
    else: horiz_angles = []

    return Angles(horiz_angles,vert_angles)

def calculate_x(r,theta,y):
```

```
    return -np.tan(theta)*(y-r*np.sin(theta))+r*np.cos(theta)

def calculate_y(r,theta,x):
    return -1/np.tan(theta)*(x-r*np.cos(theta))+r*np.sin(theta)

def get_angles(anti_img,save_params):
    flipped = np.flip(anti_img,1)
    accumulator, angles, dists =
        hough_line(flipped,np.arange(math.radians(-110),math.radians(30),.005))

    possible_lines = []

    thickness = .35

    x0,y0 = flipped.shape
    x0 /= 2
    y0 /= 2
    size = min(flipped.shape)

    for s, angle, dist in zip(*hough_line_peaks(accumulator,
          angles, dists,threshold=0*np.max(accumulator),min_distance=3)):
        x = partial(calculate_x,dist,angle)
        y = partial(calculate_y,dist,angle)

        points = [(0,y(0)),(x(flipped.shape[1]),flipped.shape[1]),
                  (x(0),0),(flipped.shape[0],y(flipped.shape[0]))]
        point1 = None
        point2 = None

        for i in range(4):
            point = points[i]
            if point[0] >= -1 and point[0] <= flipped.shape[0]+1 and
               point[1] >= -1 and point[1] <= flipped.shape[1]+1:
                if point1 is None:
                    point1 = point
                elif euclidean(point1,point) > 10:
                    point2 = point
                    break
        if point1 is None or point2 is None: continue
        template = create_template_line_two_points(*flipped.shape,
                                                   point1,point2,thickness)
        score = np.sum(template*flipped)

        possible_lines.append((score,point1,point2))

```

```
    thresh = max([x[0] for x in possible_lines])*.5
    temp_img = np.copy(flipped)
    clean_img = np.zeros(flipped.shape)
    lines = []
    while len(possible_lines) > 0:
        score,point1,point2 = possible_lines.pop(0)
        if len(lines) > 2 and score < thresh: break

        dy = point2[1]-point1[1]
        dx = point2[0]-point1[0]

        D = abs(dy*x0-dx*y0+point2[0]*point1[1]-point2[1]*point1[0])
        D /= math.sqrt(dy**2 + dx**2)
        D /= size
        if D < .25: lines.append((point1,point2))

        template = create_template_line_two_points(*flipped.shape,
                                                   point1,point2,thickness)

        just_line_img = flipped*template
        clean_img += just_line_img
        temp_img -= just_line_img
        temp_img[temp_img < 0] = 0

        temp = []
        for line2 in possible_lines:
            s,p1,p2 = line2
            template2 = create_template_line_two_points(*flipped.shape,
                                                        p1,p2,thickness)
            score = np.sum(template2*temp_img)
            if score > s*.3:
                temp.append((s,p1,p2))

        possible_lines = temp

    angles = []
    for line in lines:
        point1, point2 = line
        angle = -1*np.rad2deg(np.arctan2(point2[1]-point1[1],point2[0]-point1[0]))
        if angle > 70: angle -= 180
        angles.append(angle)
    angles = group_angles(angles)

    return angles
```

E. HOUGH ANTICROSSING TRANSFORM

This section contains the code for the Hough anticrossing transform described in section 1.2.3.

```
def get_pixel_lists(anti_img,middle):
    white_pixels = np.transpose(np.nonzero(anti_img))
    white_pixels = np.array([[x[1],x[0]] for x in white_pixels])

    xmin = (.5-middle/2)*anti_img.shape[0]
    xmax = (.5+middle/2)*anti_img.shape[0]
    ymin = (.5-middle/2)*anti_img.shape[1]
    ymax = (.5+middle/2)*anti_img.shape[1]
    candidate_pixels = []
    for pixel in white_pixels:
        if pixel[0] > xmin and pixel[0] < xmax and
           pixel[1] > ymin and pixel[1] < ymax:
            candidate_pixels.append(pixel)
    candidate_pixels = np.array(candidate_pixels)

    return white_pixels,candidate_pixels

def angle_matches(theta,angle_list,tolerance):
    best = []

    for i in range(len(angle_list)):
        if abs(theta-angle_list[i]) < tolerance:
            best.append(i)

    return best

def hough_anticrossing(white_pixels,candidate_pixels,angles,
                       anticrossing_length,params):
    down_left_accum = np.zeros((len(candidate_pixels),len(angles.horiz_angles)))
    up_right_accum = np.zeros((len(candidate_pixels),len(angles.horiz_angles)))
    up_left_accum = np.zeros((len(candidate_pixels),len(angles.vert_angles)))
    down_right_accum = np.zeros((len(candidate_pixels),len(angles.vert_angles)))

    for p1 in white_pixels:
        for j in range(len(candidate_pixels)):
            p2 = candidate_pixels[j]
            if np.array_equal(p1,p2): continue
            dist = euclidean(p1,p2)
            if dist > anticrossing_length or dist < params.min_dist_to_vote:
                continue
            if p2[0] == p1[0]: theta = -90
```

```
            else: theta = math.degrees(math.atan((p2[1]-p1[1])/(p2[0]-p1[0])))

            horiz = theta > angles.split_angle

            if horiz: best = angle_matches(theta,angles.horiz_angles,
                                           params.match_angle_tolerance)
            else: best = angle_matches(theta,angles.vert_angles,
                                           params.match_angle_tolerance)
            if len(best) == 0: continue

            right = p1[0] > p2[0]

            if horiz:
                if right: for i in best: up_right_accum[j,i] += 1
                else: for i in best: down_left_accum[j,i] += 1
            else:
                if p1[0] == p2[0]:
                    if p1[1] < p2[1]: for i in best: down_right_accum[j,i] += 1
                    else: for i in best: up_left_accum[j,i] += 1
                elif right: for i in best: down_right_accum[j,i] += 1
                else: for i in best: up_left_accum[j,i] += 1

    return down_left_accum,up_right_accum,down_right_accum,up_left_accum

def accum_angle(white_pixels,vert_accum,horiz_accum):
    num_vert = vert_accum.shape[1]
    num_horiz = horiz_accum.shape[1]

    accum = np.zeros((len(white_pixels),num_vert,num_horiz))
    for p in range(len(white_pixels)):
        for tv in range(num_vert):
            accum[p,tv,:] += vert_accum[p,tv]
    for p in range(len(white_pixels)):
        for th in range(num_horiz):
            accum[p,:,th] += horiz_accum[p,th]

    return accum
```

F. ANTICROSSING SELECTION

This section contains the code for selecting anticrossings from the Hough accumulators as described in section 1.2.4.

```
def get_points(accum,white_pixels,angles,params):
    points = []
    accum = accum.copy()
    try: max_v = np.max(accum)
    except: return points
    poss = list(range(len(white_pixels)))
    while accum.any():
        best = np.unravel_index(np.argmax(accum, axis=None), accum.shape)

        score = accum[best]/max_v
        if score < params.quality: break
        p_idx = best[0]
        accum[p_idx,:,:] = 0
        p = white_pixels[p_idx]

        theta_vert = angles.vert_angles[best[1]]
        theta_horiz = angles.horiz_angles[best[2]]
        points.append((p,theta_vert,theta_horiz,score))

        to_remove = []
        for p2_idx in poss:
            p2 = white_pixels[p2_idx]
            if euclidean(p,p2) < params.neighborhood:
                accum[p2_idx,:,:] = 0
                to_remove.append(p2_idx)
        for x in to_remove: poss.remove(x)

    return points

def find_anticrossings(anti_img,white_pixels,angles,down_accum,up_accum,
                       params,aa_params,multi):
    down_points = get_points(down_accum,white_pixels,angles,params)
    up_points = get_points(up_accum,white_pixels,angles,params)
    log.debug(f'down points: {down_points}')
    log.debug(f'up points: {up_points}')

    poss = []

    for i in range(len(down_points)):
        p1,v1,h1,s1 = down_points[i]
        for j in range(len(up_points)):
            p2,v2,h2,s2 = up_points[j]

            if tuple(p2) == tuple(p1): continue
            if euclidean(p1,p2) > params.anticrossing_width_max or
```

```
                euclidean(p1,p2) < params.anticrossing_width_min: continue
            if abs(v1-v2) > params.same_angle_tolerance or
                abs(h1-h2) > params.same_angle_tolerance: continue

            theta = math.degrees(math.atan2(p2[1]-p1[1],p2[0]-p1[0]))
            if abs(v1+180-theta) < aa_params.flat_buffer or
                abs(theta-h1) < aa_params.flat_buffer: continue
            if theta > v1+aa_params.between_buffer and
              theta < h1+180-aa_params.between_buffer:
                v = np.average([v1,v2],None,[s1,s2])
                h = np.average([h1,h2],None,[s1,s2])
                ans = Anticrossing(tuple(p1),(h,v),tuple(p2),(h,v))
                template = create_template_anticrossing(*anti_img.shape,
                                        ans,aa_params.anticrossing_length,.5)

                s = np.sum(template*anti_img)
                poss.append((s,ans))

    poss.sort(key=lambda x: x[0], reverse=True)
    if len(poss) == 0: return poss
    if not multi: return [poss[0][1]]

    ans = []
    s_thresh = poss[0][0]*params.quality
    dist_thresh = aa_params.anticrossing_length*.6

    while len(poss) > 0:
        s, current = poss.pop(0)
        if s < s_thresh: break
        ans.append(current)
        to_remove = []
        for pair in poss:
            other = pair[1]
            dists = [euclidean(current.lower.point,other.lower.point),
                     euclidean(current.lower.point,other.upper.point),
                     euclidean(current.upper.point,other.lower.point),
                     euclidean(current.upper.point,other.upper.point)]
            if min(dists) < dist_thresh: to_remove.append(pair)
        for pair in to_remove: poss.remove(pair)

    return ans
```

G. OPTIMIZATION

This section contains the code for the optimization algorithm described in section 1.2.5.

```
def even_distribution(start,stop,num_steps):
    step_size = (stop-start)/num_steps
    return np.arange(start,stop+step_size,step_size)

def score_angle(anti_img,point,angle,length,thickness):
    if angle < -110 or (angle > 20 and angle < 70) or (angle > 200): return 0
    xsize,ysize = anti_img.shape
    template = create_template_line_full_image(xsize,ysize,point,angle,
                                               length,thickness,False)
    size = sum(sum(template))/255.
    if size == 0: return 0
    s = sum(sum(template*anti_img))
    return s

def search_for_best_angle_at_point(anti_img,point,start,stop,num_steps,length,
                                   thickness,cur_best_score,cur_best_angle):
    score = cur_best_score
    best_angle = cur_best_angle
    scores = []

    for angle in even_distribution(start,stop,num_steps):
        s = score_angle(anti_img,point,angle,length,thickness)
        scores.append((angle,s))
        if s > score:
            best_angle = angle
            score = s
        if s < score*.7: break

    if score <= cur_best_score: return score, best_angle

    scores = [ x for x in scores if x[1]>score*.95 ]
    total_score = sum([x[1] for x in scores])
    ans = sum([x[0]*x[1] for x in scores])/total_score

    return score, ans

def optimize_angle(anti_img,point,start_angle,anticrossing_length,params):
    best_angle = start_angle
    score = 0

    score, best_angle = search_for_best_angle_at_point(anti_img,point,start_angle,
```

```
                            start_angle+params.max_angle_change,params.num_angle_steps,
                            anticrossing_length,params.thickness,score,best_angle)
    score, best_angle = search_for_best_angle_at_point(anti_img,point,start_angle,
                           start_angle-params.max_angle_change,params.num_angle_steps,
                           anticrossing_length,params.thickness,score,best_angle)

    return score,best_angle

def score_point(anti_img,point,angles,anticrossing_length,params):
    score1,angle1 = optimize_angle(anti_img,point,angles[0],
                                   anticrossing_length,params)
    score2,angle2 = optimize_angle(anti_img,point,angles[1],
                                   anticrossing_length,params)
    return score1,score2,angle1,angle2

def next_points(point,angle):
    if abs(angle) > 45 and abs(angle) < 135:
        return [(point[0],point[1]+1),(point[0],point[1]-1)]
    else:
        return [(point[0]-1,point[1]),(point[0]+1,point[1])]

def find_next_points(point,angle1,score1,angle2,score2):
    ans = []
    for p in next_points(point,angle1): ans.append((p,score1))
    for p in next_points(point,angle2): ans.append((p,score2))
    return ans

def check_and_add_points(frontier,explored,xmax,ymax,points_to_add,
                         starting_point,max_point_dist):
    frontier_points = [x[0] for x in frontier]
    for pair in points_to_add:
        point = pair[0]
        valid = point not in explored and point not in frontier_points
        valid = valid and euclidean(point,starting_point) < max_point_dist
        valid = valid and point[0] >= 0 and point[0] < xmax
        valid = valid and point[1] >= 0 and point[1] < ymax
        if valid: frontier.insert(0,pair)
    frontier.sort(key=lambda x: x[1],reverse=True)

def optimize_triple_point(anti_img,triple_point,anticrossing_length,params):
    score1 = 0
    score2 = 0
    best_point = triple_point.point
    best_angles = (triple_point.left(),triple_point.right())

```

```
    frontier = [(triple_point.point,0),]
    explored = []

    ymax,xmax = anti_img.shape

    while len(frontier) > 0:
        cur_point,_ = frontier.pop(0)
        explored.append(cur_point)
        s1,s2,angle1,angle2 = score_point(anti_img,cur_point,best_angles,
                                          anticrossing_length,params)

        if s1 + s2 > score1+score2:
            score1 = s1
            score2 = s2
            best_point = cur_point
            best_angles = (angle1,angle2)

        if len(explored) >= params.num_to_search: break

        if s1 < .8*score1 and s2 < .8*score2: continue

        next_points = find_next_points(cur_point,angle1,s1,angle2,s2)
        check_and_add_points(frontier,explored,xmax,ymax,next_points,
                             triple_point.point,params.max_point_dist)

    return best_point, best_angles, score1, score2

def optimize_anticrossings(anti_img,anticrossings,aa_params,params,
                           match_angles=False):
    if params.num_to_search == 0: return anticrossings
    optimized_anticrossings = []
    for ac in anticrossings:
        point1,angles1,score11,score12 = optimize_triple_point(anti_img,
                               ac.lower,aa_params.anticrossing_length,params)
        point2,angles2,score21,score22 = optimize_triple_point(anti_img,
                               ac.upper,aa_params.anticrossing_length,params)

        if abs((angles1[1]+360-angles1[0])-180) < aa_params.flat_buffer or
           abs((angles2[0]-angles2[1])-180) < aa_params.flat_buffer:
            continue

        if match_angles:
            h = np.average([angles1[0]-180,angles2[1]],None,[score11,score22])
            v = np.average([angles1[1],angles2[0]-180],None,[score12,score21])
            angles1 = (h+180,v)
```

```
            angles2 = (v+180,h)
        else:
            if score11 > 3*score22: angles2 = (angles2[0],angles1[0]-180)
            if score22 > 3*score11: angles1 = (angles2[1]+180,angles2[1])
            if score12 > 3*score21: angles2 = (angles1[1]+180,angles2[1])
            if score21 > 3*score12: angles1 = (angles1[0],angles2[0]-180)

        line = Anticrossing(point1,(angles1[0]-180,angles1[1]),
                            point2,(angles2[1],angles2[0]-180))
        optimized_anticrossings.append(line)

    return optimized_anticrossings
```

H. COMPLETE ALGORITHM

Finally, this section contains the code that executes the complete algorithm.

```
def analyze_with_hough(data,binary_params=None,optimization_params=None,
                       aa_params=None, hough_params=None,pair_params=None,
                       multi=False):

    if binary_params is None: binary_params = BinaryParameters()
    if optimization_params is None: optimization_params = OptimizationParameters()
    if aa_params is None: aa_params = AShapeParameters()
    if hough_params is None: hough_params = HoughParameters()
    if pair_params is None: pair_params = APairParameters()

    anti_img = make_binary(data,run_params.data_name,binary_params)
    aa_params.anticrossing_length =
                           aa_params.anticrossing_length*min(anti_img.shape)
    pair_params.anticrossing_width_max =
                           pair_params.anticrossing_width_max*min(anti_img.shape)
    white_pixels,candidate_pixels = get_pixel_lists(anti_img,hough_params.middle)

    angles = get_angles(anti_img,save_params)
    down_left_accum,up_right_accum,down_right_accum,up_left_accum =
            hough_anticrossing(white_pixels,candidate_pixels,angles,
                               aa_params.anticrossing_length,hough_params)
    down_accum = accum_angle(candidate_pixels,down_right_accum,down_left_accum)
    up_accum = accum_angle(candidate_pixels,up_left_accum,up_right_accum)

    anticrossings =  find_anticrossings(anti_img,candidate_pixels,angles,
                        down_accum,up_accum,pair_params,aa_params,multi)
```

```
27      anticrossings = optimize_anticrossings(anti_img,anticrossings,aa_params,
28                          optimization_params,match_angles=True)
29      anticrossings = optimize_anticrossings(anti_img,anticrossings,aa_params,
30                          OptimizationParameters.small_optimization())
31
32      #specific to data format so excluded
33      converted = convert_to_data_space(anticrossings)
34      return converted
```

B. PARAMETER SETTINGS FOR ANTICROSSING DETECTION

One of the downsides of the algorithm presented here for anticrossing detection is that it is reliant on setting many parameters. We have found that these parameters need to be set on a device specific, rather than on a scan specific basis. The parameters in the algorithm can be divided into two categories – those that describe the shape of the anticrossing and those that describe how the algorithm should operate. What follows is a list of the parameters and what we have found to be reasonable defaults for them.

Parameters that describe the shape of the anticrossing:

- anticrossing length: the percentage of the scan each leg of the anticrossing takes up
  - .4 for single mode
  - .15 for multi mode
- flat buffer: number of degrees the angle between the two legs should be from 180 degrees
  - 10 degrees
- between buffer: minimum angle (in degrees) between the connecting line and a leg of the anticrossing
  - 30 degrees for single mode.
  - 40 degrees for multi mode
- anticrossing width min: the minimum distance between the two triple points, in percentage of scan
  - 0
- anticrossing width max: the maximum distance between the two triple points, in percentage of scan
  - .5 for single mode
  - .15 for multi mode
- same angle tolerance: how many degrees different can the two inclinations of the two triple points can be to pair into an anticrossing
  - 25 degrees

Parameters that describe how the algorithm should run:

- Binary Formation
  - grad percentile: what percentile of the gradient to set as the threshold for peaks
    - .95
  - border: the number of pixels to exclude from the edges in forming the binary image
    - 5 pixels
  - peak width: the number of pixels on either side that a peak must be bigger than
    - 5 pixels
  - remove switches: whether or not to run the switch removal process
    - device specific
  - switch perc: the percentage of a row that must be white to count as a switch
    - .6
  - white col perc: the percentage of a column that must be white to not be removed as part of a switch
    - .1
  - isolated thresh: the number of pixels in the 3×3 grid must be white to not be removed
    - 2 pixels
  - binary closing iterations: the number of iterations to run of the binary closing algorithm
    - device specific
  - straighten lines: whether or not to run the line straightening process
    - True
- Hough Anticrossing Parameters
  - min distance to vote: how far away two pixels must be to vote (as close pixels have greater error in determining the inclination between them)
    - 4 pixels (but very small scans should probably drop this number)
  - match angle tolerance: how close the inclination between two pixels must match one of the detected inclinations to vote
    - 2 degrees
  - middle: what percentage of the middle of the image is considered as candidate triple points
    - .5
- Filter Anticrossing Parameters

    - neighborhood: the minumum distance between two possible triple point candidates, in pixels
        * 5 pixels
    - quality: the percentage of the max scoring triple point parameters a candidate must have
        * .5
- Optimization Parameters
    - max angle change: the maximum number of degrees an inclination can change in the optimization
        * 45 degrees
    - num angle steps: the number of inclinations to search at a given point to try in the optimization
        * 60
    - max point change: the number of pixels a triple point can move in the optimization
        * 10 pixels
    - num to search: the maximum number of pixel locations to search for each triple point in the optimization
        * 50

If you are not getting the desired results, then determining if the anticrossings match the parameter specifications or which step of the algorithm is failing, and adjusting those parameters, is a good first step for improving them.

DISTRIBUTION

Hardcopy—External

Number of Copies	Name(s)	Company Name and Company Mailing Address

Hardcopy—Internal

Number of Copies	Name	Org.	Mailstop
1	D. Chavez, LDRD Office	1911	0359

Email—Internal (encrypt for OUO)

Name	Org.	Sandia Email Address
Technical Library	01177	libref@sandia.gov

Sandia National Laboratories is a multimission laboratory managed and operated by National Technology & Engineering Solutions of Sandia LLC, a wholly owned subsidiary of Honeywell International Inc., for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-NA0003525.