UNSW
AUSTRALIA

SCHOOL OF ELECTRICAL ENGINEERING
AND TELECOMMUNICATIONS

Automated Tracking of Single-Atom Qubits for Silicon Quantum Computing

by

Dennis Otter

Student ID: 5059446

Thesis submitted as a requirement for the degree

Bachelor of Engineering in Electrical Engineering

Submitted: November 2, 2018
Supervisor: Andrea Morello

Abstract

Promising research is being done at the Centre for Quantum Computation and Communication Technology (CQC2T) investigating quantum bits constructed from phosphorus donor atoms hosted in silicon semiconductor devices. These single-atom quantum devices aim to read and control a phosphorus donor electron spin as a qubit. Each time these devices are cooled to $\sim 15mK$, researchers must manually scan for the donor atoms with voltage gates. Doing so causes transitions, where electrons tunnel off donors, which can be detected and used to determine the capacitive coupling of each gate to the donors.

The aim of this thesis is to automatically identify and track these transitions. This in turn will reveal the capacitive coupling of voltage gates to donor atoms, which will reduce manual labour, and has important applications in donor triangulation, automated tuning, and donor coupling.

Acknowledgements

I would like to thank Andrea Morello, my supervisor, for giving me the opportunity to pursue a thesis topic in quantum engineering. This is where my interests lie and I really enjoy being able to contribute to such an exciting area of research!

I would also like to sincerely thank Serwan Assad for all his help and support. Serwan has met with me almost every week over the course of Thesis A and B, and spent countless hours providing me with an incredible amount of information and guidance. An additional special thanks goes to Serwan for his patience with my unsolicited consultations.

Thanks to Mark Johnson for his general advice, running scans for me, and for his help testing the transition motion tracking component.

Thanks to Hannes Firgau for the plethora of advice, and for the valuable comments on my poster presentation.

A most special thanks, goes to my partner Abigail Standish. For all the advice and guidance. For bearing with my esoteric explanations of my thesis. For always unconditionally loving and supporting me.

Abbreviations

CQC2T Centre for Quantum Computation and Communication Technology

SET Single Electron Transistor

FET Field Effect Transistor

2D 2-Dimensional

3D 3-Dimensional

Contents

List of Figures

Chapter 1

Introduction

1.1 Problem Definition

Researchers at CQC2T are investigating the readout and control of an electron spin as a qubit. Currently, the single-atom quantum devices must be manually characterised each time they are cooled to operating temperature $\sim 15mK$. This poses a problem for researchers, as they must spend a lot of time locating donor atoms and determining the capacitive coupling of each gate to the donors. Recent devices are comprised of seven gates, which must all be set to specific voltages in order for correct tuning. Additionally, if a device is already tuned and one gate is modified, the others need to be adjusted to accomodate. It is desirable to determine how each gate affects the devices, so that gates can be compensated when changing values.

1.2 Thesis Objectives

To address these obstacles posed to researchers, a system must be developed that can automatically identify transitions, and track how the voltage gates affect them. Thus the system should ultimately be capable of:

1. Finding Transitions within a charge stability diagram.
2. Track how a transition moves within a charge stability diagram, as a third gate is modified.

Chapter 2

Background

2.1 Why Quantum Devices: Limitations of Classical Computers

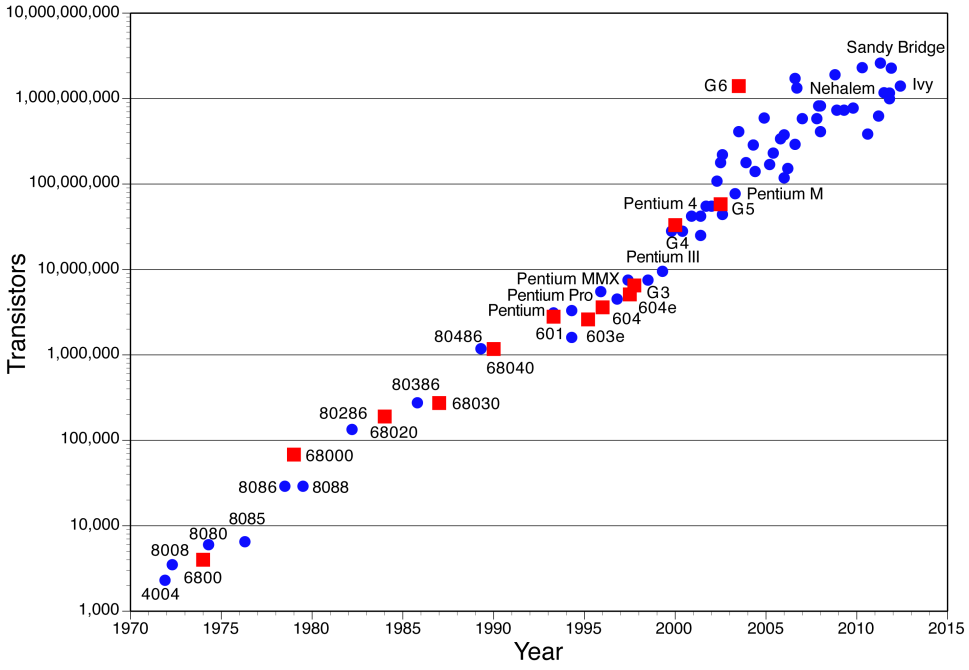


Figure 2.1: Moore's law - transistors on integrated circuit chips (1970-2015). Image from Our World in Data [1].

In 1965 Gordon Moore, co-founder of Intel and Fairchild Semiconductor, predicted the number of transistors on a single chip would double every two years [6]. To date, this has been accurate, with commercial gate sizes shrinking to the 7 nm node as of May 2018 [7]. With components continuing to miniaturise towards the scale of single atoms, quantum mechanical

phenomena will begin to take effect. The Bohr radius of a phosphorus atom in silicon is 2.5 nm, not much smaller than current transistor sizes.

2.2 Qubits (Quantum Bits)

Fortunately, work is being done to develop new technologies that exploit the laws of quantum mechanics. One such device is a "quantum computer" that operates based on quantum bits, or qubits [8]. Theoretically, a qubit can be any two-level quantum system with eigenstates $|1\rangle, |0\rangle$ that are well separated in energy. While classical bits may be either 0 or 1, a qubit may exist in a superposition state:

$$|\psi\rangle = \alpha|1\rangle + \beta|0\rangle = \begin{pmatrix} \alpha \\ \beta \end{pmatrix} \quad \text{where} \quad |\alpha|^2 + |\beta|^2 = 1 \quad \alpha, \beta \in \mathbf{C}$$

2.3 Quantum Entanglement

Additionally, multiple qubits can be entangled to form states that cannot be constructed from individual base states. A non-entangled two-qubit system can be described as the tensor product of its two base qubits. From this, the base states of a two-level system are:

$$|0\rangle \otimes |0\rangle = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} \quad |0\rangle \otimes |1\rangle = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} \quad |1\rangle \otimes |0\rangle = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} \quad |1\rangle \otimes |1\rangle = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}$$

However, qubits can be entangled, which allows the system to exist in superposition states that cannot be described as the product of two seperate states [9]. For example, the Bell state is a valid maximally entangled state that cannot be expressed as the tensor product of two individual qubits:

$$|\psi\rangle = \frac{1}{\sqrt{2}}(|1\rangle \otimes |1\rangle + |0\rangle \otimes |0\rangle) = \frac{1}{\sqrt{2}} \begin{pmatrix} 1 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

These properties of qubits allow quantum computers to achieve exponential speedup over classical computers [10].

2.4 Electron Spin

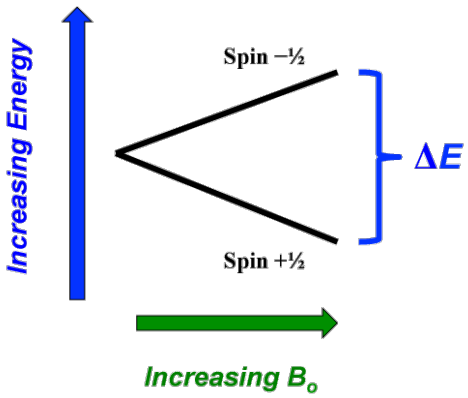


Figure 2.2: Zeeman splitting. Stronger magnetic field relates to larger energy splitting. Image from Questions and Answers in Magnetic Resonance Imaging [2].

In quantum mechanics, electrons have an intrinsic angular momentum known as spin. Spin is intrinsic to other particles such as hadrons, atomic nuclei, and other fermions. This thesis will deal with spin in relation to electrons. Spin gives rise to a magnetic moment, which in electrons is described by $-g_e \mu_B S$. $-g_e$ is the electron spin g-factor, μ_B is the Bohr magneton, and S is the electron spin. As electrons are fermions, they have half-integer spin; $S = \pm \frac{\hbar}{2}$.

In the presence of a magnetic field, a magnetic moment parallel with the field will have a distinct energy difference from an anti-parallel moment. Electron spins exhibit this with two distinct energy states, separated by the Zeeman splitting $E_Z = g\mu_B B$. As the magnetic field increases, so does the energy splitting between spin up and down, as illustrated in Figure 2.2. Zeeman splitting gives rise to two base states; spin-up $|\uparrow\rangle$ and spin-down $|\downarrow\rangle$. Interestingly, an electron's spin can exist in a superposition state, where it can be both spin-up and spin-down. Thus an electron spin can be used as a qubit [11], with the spin-up and spin-down states forming the basis for a two-level quantum system.

The spin-up and spin-down states can be written as:

$$|\uparrow\rangle = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad \text{and} \quad |\downarrow\rangle = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

As such, an electron's spin can be expressed as any superposition state of the form:

$$|\psi\rangle = \alpha\,|\uparrow\rangle + \beta\,|\downarrow\rangle = \begin{pmatrix} \alpha \\ \beta \end{pmatrix} \quad \text{where} \quad |\alpha|^2 + |\beta|^2 = 1 \quad \alpha, \beta \in \mathbf{C}$$

2.5 Quantum Tunnelling

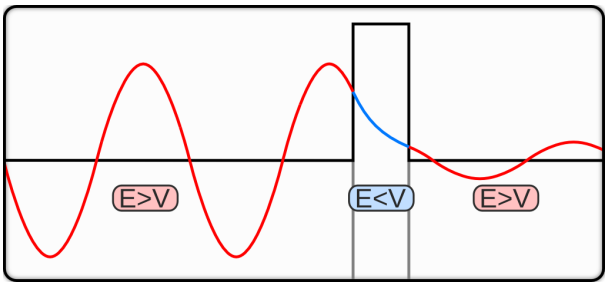


Figure 2.3: Quantum tunnelling through a barrier. Image from Felix Kling [3].

Quantum tunnelling is a quantum mechanical phenomenon where a particle may tunnel through a barrier that is classically insurmountable. In classical physics, the location of a particle or object may be clearly identified. However, this is not applicable in quantum systems, where particles experience wave-particle duality. One implication of wave-particle duality is that the location of a particle can be represented as a probability wave-function [12].

The location of an electron is described by a probability wave-function. When presented with a potential barrier, a portion of this wave function can exist past the barrier, meaning there is a small probability that the electron can tunnel through it. This probability is influenced by the width and intensity of the potential barrier. This is shown in Figure 2.3 which depicts the wave function of a particle near a potential barrier that exceeds the particle's energy. It can be seen that the function does extend past the barrier, albeit with a decreased probability magnitude.

2.6 Single Electron Transistor

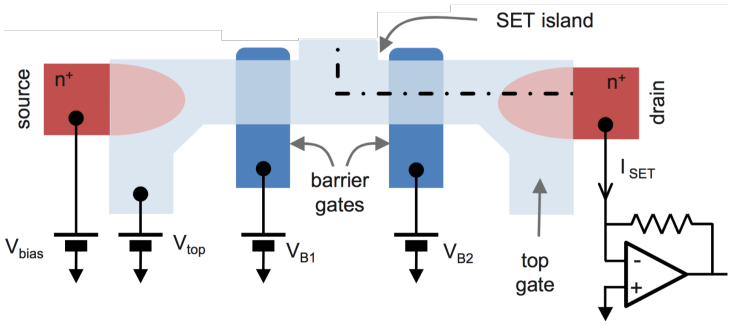
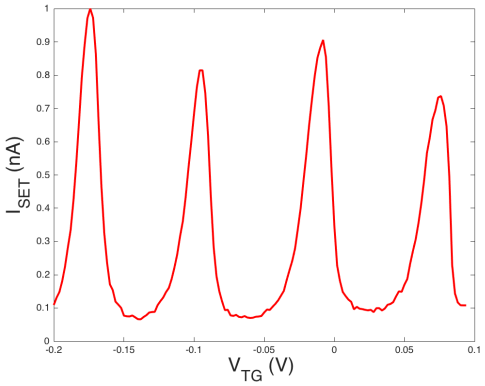
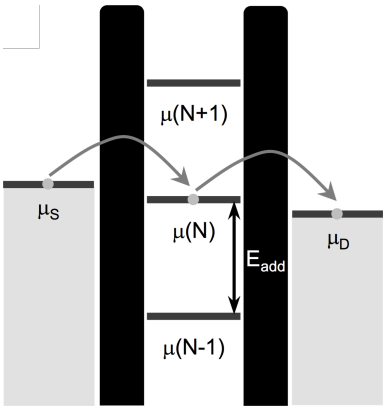


Figure 2.4: Visualisation of a SET. Image from Morello et al. study [4].



(a) Periodic coulomb peaks. Image generated in MatLab from sample data.



(b) Electrochemical potential diagram. Image from Jarryd Pla PhD [13].

Figure 2.5: SET island conduction patterns.

A single electron transistor (SET) aims to have the same capability as a normal transistor; controlled current flow under certain conditions. A basic SET consists of a drain, island, source, and two barrier gates [14]. The drain, source, and two barrier gates are coupled to voltage references, such that their potential can be controlled (as in Figure 2.4. The island is an electron reserve between the source and drain. As opposed to a n-channel field effect transistor (FET) that controls current using the gate-drain voltage, a SET controls current based on the electrochemical potential of the island.

When the voltages of the source, drain, and barriers are set appropriately, electrons may tunnel from the source to the island, then from the island to the drain, forming a current as

shown by the arrows in Figure 2.5b. If the island's level ($\mu(N)$) is below the source (μ_S), an electron is able to tunnel from the source to the island ($\mu_S > \mu(N)$ in Figure 2.5b). If the island's level is above the drain (μ_D), an electron is then able to tunnel from the island to the drain ($\mu_D < \mu(N)$ in Figure 2.5b). If both of these conditions are true, electrons are able to tunnel from source, to island, to drain. This forms an electrical current which can be measured using traditional electronics, as shown by I_{SET} in Figure 2.4.

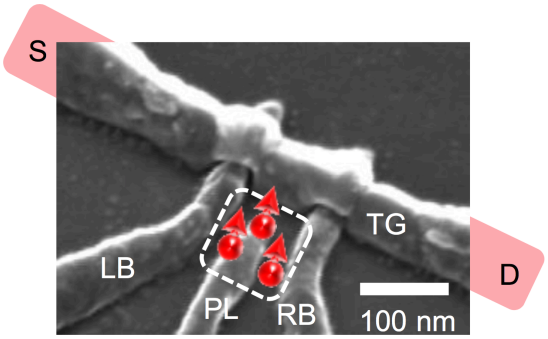
The barrier gate voltages may be adjusted to allow more or less current to flow. An ideal barrier gate voltage would enable maximal current flow in the conduction region and no current otherwise.

The electrochemical potential of the island is affected by how many electrons are on the island, and the surrounding electrostatic landscape. Many factors influence the electrostatic landscape, such as specific gates designed to control the level of the island or nearby ionised atoms.

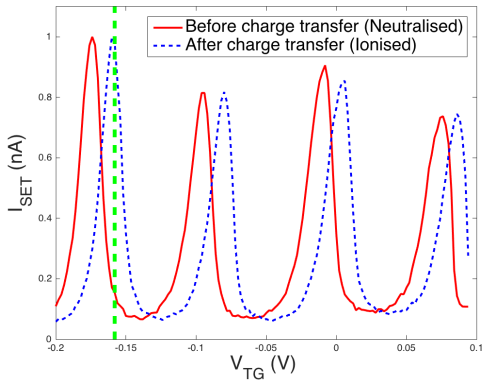
Consider an island will be in the conduction region when it has N electrons. Once an electron tunnels onto the island, it will have N+1 electrons. Once the electron tunnels off, the island will have N electrons again. If the island's potential is raised to the point where it is in the conduction region with N-1 electrons, then it will go from N-1, to N, then back to N-1 electrons as the tunnelling process occurs. A similar process will occur if the island's potential is lowered to being in the conduction region with N+1 electrons. Thus, as the island's potential is increased or decreased, the SET will traverse periodic regions of conductivity [13]. A top-gate is employed to influence the potential of the island. Evaluating the SET current with respect to the top-gate voltage reveals periodic coulomb peaks. These can be seen in in Figure 2.5a which depicts I_{SET} experiencing current peaks as the top gate voltage is varied.

The other major influence on the island's potential is nearby donor atoms.

2.7 Donor Atoms



(a) Donor implant region near the SET island. Image from Jarryd Pla PhD [13].



(b) Before/after comparison of shifted coulomb peaks due to a charge transfer event. Image generated in MatLab from sample data.

Figure 2.6: Donor Atoms near the SET island.

In the single-atom quantum devices, phosphorus atoms (^{31}P) are implanted in a silicon (^{28}Si) substrate near the SET island (See Figure 2.6a) to act as donor atoms. ^{28}Si is used as it is nuclear spin-zero, providing very little magnetic background disturbance. ^{31}P atoms have one free valence electron which can be operated as a qubit [15].

When a valence electron is on a donor atom, the atom is neutralised. When this valence electron is removed, the atom becomes positively ionised, influencing the surrounding electrostatic landscape, thus influencing the SET island. A nearby ionised donor atom will increase the island's potential, whereas a neutralised atom will not influence the island. Evaluating the SET current against the top-gate voltage with an ionised donor nearby will yield a similar periodic conduction pattern to prior, however these peaks will be shifted due to the donor's influence as shown in Figure 2.6b. It is possible to tune the top-gate such that the SET will conduct when a donor is ionised, and will not conduct when the donor is neutralised. Figure 2.6b displays a **green** line at such a point, where I_{SET} is low for a neutralised donor, and high for an ionised donor.

Additional gates are employed as donor gates, such as the PL gate in Figure 2.6a, which can raise or lower the electrostatic landscape near the donor atoms. This enables the potential energy of the donors to be altered, which in turn enables ionisation or neutralisation of donors.

2.8 Spin Dependent Tunnelling

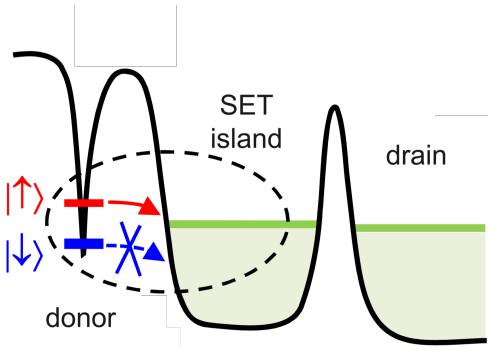


Figure 2.7: Depiction of spin dependent tunnelling. Image from Morello et al. study [4].

As mentioned previously, in the presence of a magnetic field, electrons have two distinct energy states, spin-up and spin-down, separated by Zeeman splitting $E_Z = g\mu_B B$. A spin-up electron will have a higher energy than a spin-down electron. The gates may be set such that the energy of a spin-up electron on a donor atom will be above the SET island, while the energy of a spin-down electron on the donor will be below the island. This configuration, depicted in Figure 2.7 allows a spin-up electron enough energy to tunnel off the donor, ionising it, while a spin-down electron will not have enough energy to tunnel off, keeping the atom neutral [5]. The gates can then be further tuned such that an ionised donor will cause the SET to conduct, while a neutralised donor will not. This can be seen in Figure 2.6b.

In this configuration, a spin-up electron will cause a current spike in the SET, while a spin-down electron will not, forming a system that can effectively read out the spin state of an electron. This method of using the SET as a charge detector was pioneered by the research group at CQC2T [4].

2.9 Charge Stability Diagrams

While varying both the top gate and a donor gate, the SET current can be plotted as a 2D colour plot. These plots are charge stability diagrams, with the donor gate on the x-axis, top gate on the y-axis, and SET current on the z-axis (colour). Varying the donor gate along the X axis will mainly affect the donors, while varying the top-gate along the Y axis will mainly affect the island. However, the donor gates still influence the island, and the top-gate still influences the donors.

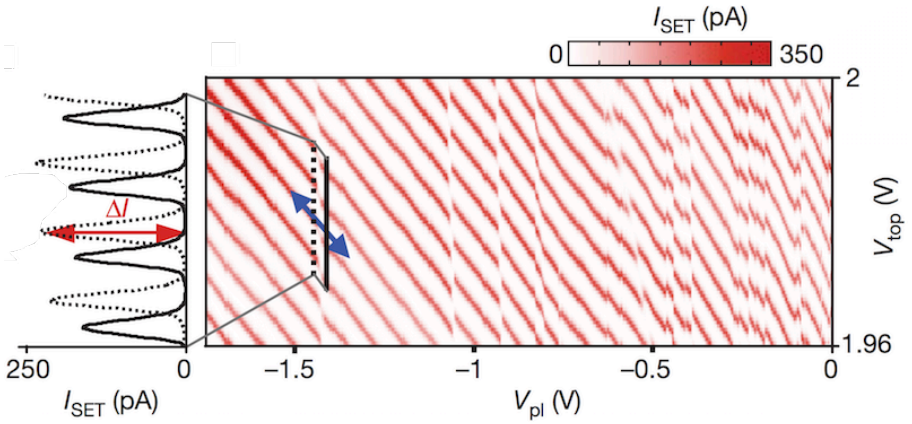


Figure 2.8: Charge stability diagram with a vertical cross section of a charge transfer event. Image from Morello et al. study [5].

A vertical cross section will yield the same periodic conduction plot as Figure 2.6b. Charge stability diagrams now show these coulomb peaks as ridges, seen in Figure 2.8. Along the x-axis, as the donor gate is varied, the coulomb peaks decline, indicating the donor gates are somewhat coupled to the island's potential. The gradient of these coulomb peaks indicates the relative capacitive coupling of the donor gate to the SET island.

As the donor gate voltages are increased or decreased, the electrochemical potential around the donor atoms is altered. At a point, this will cause the valence electron of a donor to tunnel off, ionising that atom. This is indicated by a break in a coulomb peak on a charge stability diagram [5]. The breaks in coulomb peaks form transition 'lines' which all correspond to a charge transfer event of the same donor atom. Multiple donor gates are used in different positions around the donor region to give more control over the electrostatic potential of different donors.

For spin-dependent tunnelling to occur, the device should be tuned to the positive side of a coulomb peak on a transition. In this configuration, a spin-up electron will tunnel off and bring the SET into coulomb blockade, while a spin-down electron will stay on, keeping the SET on the coulomb peak.

Additionally, the gradient of a transition will reveal the relative coupling of the donor gate and top gate to the donor atom. Thus it is highly desirable to determine the location and gradient of transitions, as this reveals the voltages necessary for spin-dependent tunnelling, and the relative coupling of the gates to the donor atoms.

2.10 Hough Transform

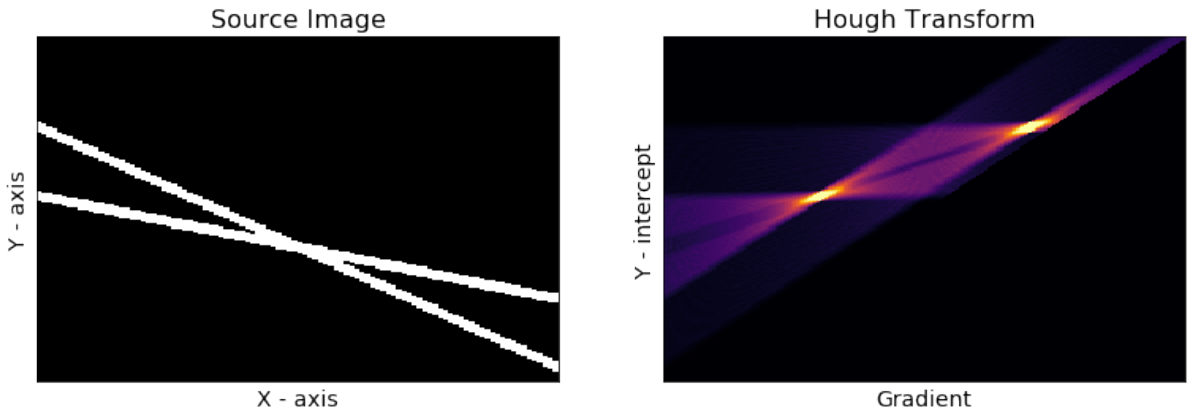


Figure 2.9: Source image and its Hough transform. Image generated in Python.

In digital image processing, the task often arises of identifying lines and other shapes. This section will cover the Hough transform as a method to achieve the task of identifying lines [16].

Lines can be characterised by two parameters m and b in the following form: $y = m \cdot x + b$. However, this form is not able to express lines that are purely vertical. Thus, often the parameters θ and r are used in the form: $r = x \cdot cos\theta + y \cdot sin\theta$. The Hough transform in this thesis uses the m and b parameters, which will be discussed in Section 3.2.3.

The Hough transform uses this definition of a line to scan the source image for lines that match possible m and b values. It does this by creating lines that span the source image, then summing the pixel values that the line intersects. The sum of the source pixels will then be stored in a matrix at the coordinates (b, m). If many of the source pixels have a high value, then the sum will be high and a peak will be shown in the results matrix. After all desired m and b values have been scanned, the resulting matrix can be scanned for peaks, which are indicative of lines detected in the source image. Figure 2.9 visualises two peaks in a Hough transform, representing the two lines from the source image.

Chapter 3

Solution

3.1 Solution Overview

Ultimately, the aim of this thesis is to develop a system that can automatically identify and track transitions. Thus, a solution must be developed that can automatically complete the following tasks:

1. Transition identification - within a given charge stability diagram, the solution should be able to identify any transitions present.

2. Transition tracking - with a stable method for identifying transitions in a charge stability diagram, the solution should be able to track how a transition moves as a third gate is modified.

A solution has been developed in Python and compiled into a library. A high level overview of the solution will be demonstrated before going into further depth. The full source code can be found in Appendix B - Solution Code or accessed at:

`https://github.com/dennisotter/D-Otter-Honours-Thesis`.

3.1.1 Identifying Transitions

A function `find_transitions()` has been written that, given a charge stability diagram, will return a list of transitions found. `find_transitions()` takes a charge stability diagram as input, along with arrays of the gate voltages on the x and y axis. The return values and arguments will be covered in detail in Section 3.2.6.

Sample case:

```
# DB   = DBL_DBR gate voltage array
# TGAC = TGAC voltage array
# DC_voltage  = Charge stability diagram with axes [DB,TGAC]
find_transitions(DB,TGAC,DC_voltage,plot=True);
```

Sample output:

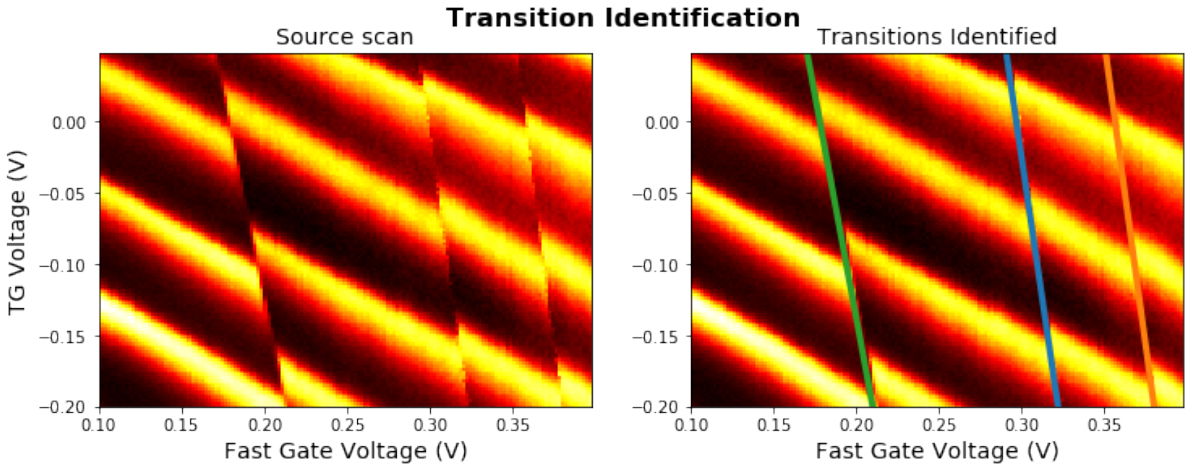


Figure 3.1: Sample transition identification. Image generated in Python.

Figure 3.1 shows the output of `find_transitions()`; a charge stability diagram and a duplicate diagram with the transitions from the function highlighted. The function is very effective at identifying transitions within a charge stability diagram and therefore successfully addresses one of the main goals of this thesis. The underlying code will be discussed in detail in Section 3.2.

3.1.2 Tracking Transitions

A function `find_transitions_3D()` has been written, that calls the function `find_transitions()` on slices of a 3-dimensional charge stability diagram. This locates transitions in each slice, then a tracking function can later correlate the 2D transitions. The exact operation of these functions will be covered in Section 3.3.

Sample Case:

```
# slow   = DFR gate voltage array
# fast   = DBL_DBR gate voltage array
# TG     = TGAC voltage array
# DC_voltage  = Charge stability diagram with axes [slow,TG,fast]
transition_list = find_transitions_3D(slow,fast,TG,DC_voltage)
tracking        = track_transitions_single(slow, fast, TG,
                                           DC_voltage, transition_list)
plot_transitions_3D(slow, fast, TG, DC_voltage,
                    transition_list, tracking, slices=True)
display(tracking)
```

Sample output:

```
[{'TG/fast gradient': -4.6285897182597706,
  'TG/slow gradient': 0.96460389629642296,
  'fast intercept': 2.1711063596862799,
  'fast/slow gradient': -4.7984356439271565}]
```

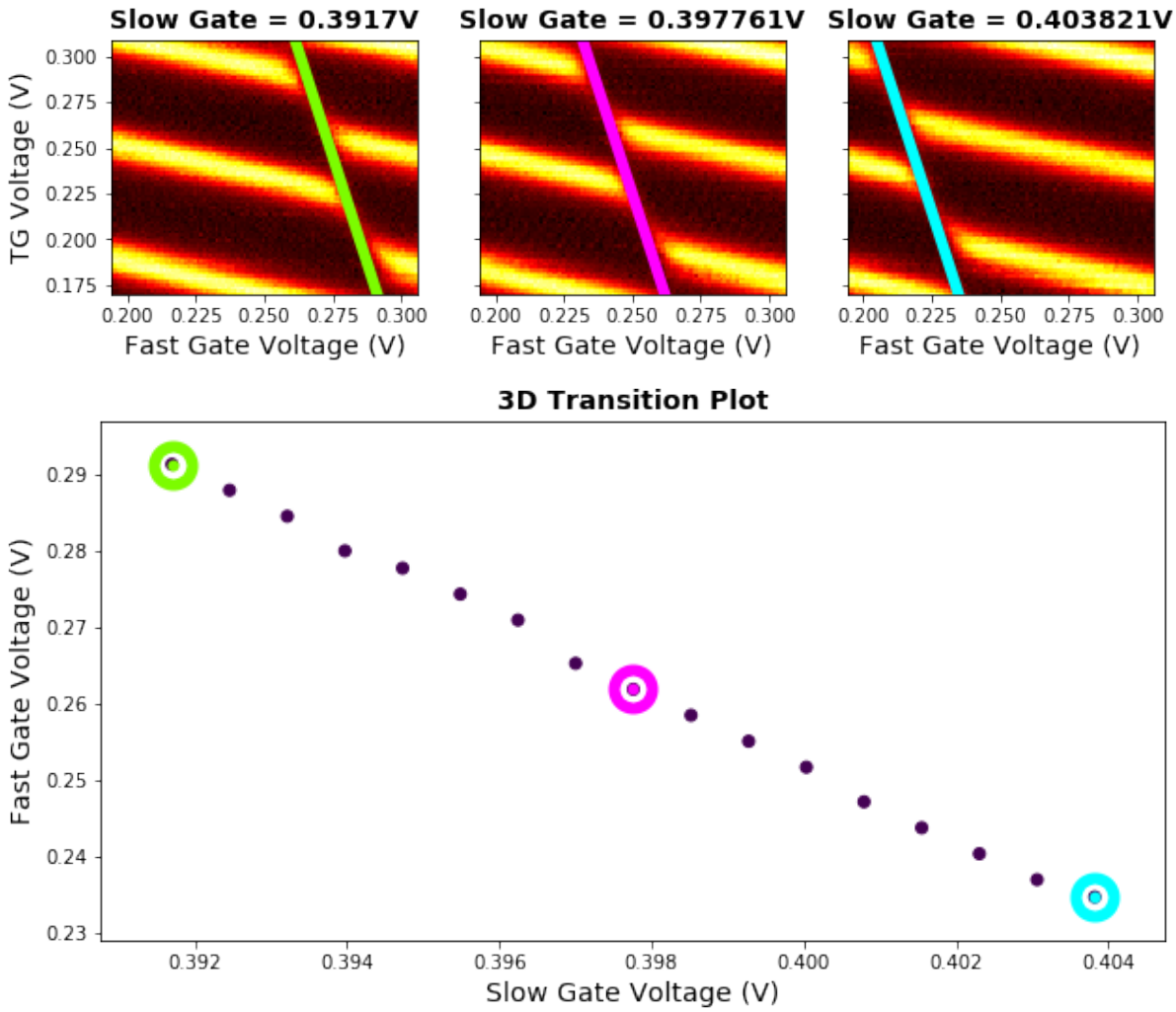


Figure 3.2: Sample single transition tracking. Image generated in Python.

As seen in Figure 3.2, the solution is capable of automatically tracking transitions over 3-dimensions. The output gradient data reveals the relative capacitive coupling of the three gates to the tracked donor. Further details will be covered in Section 3.3.

3.2 Identifying Transitions

3.2.1 Sobel Filtering

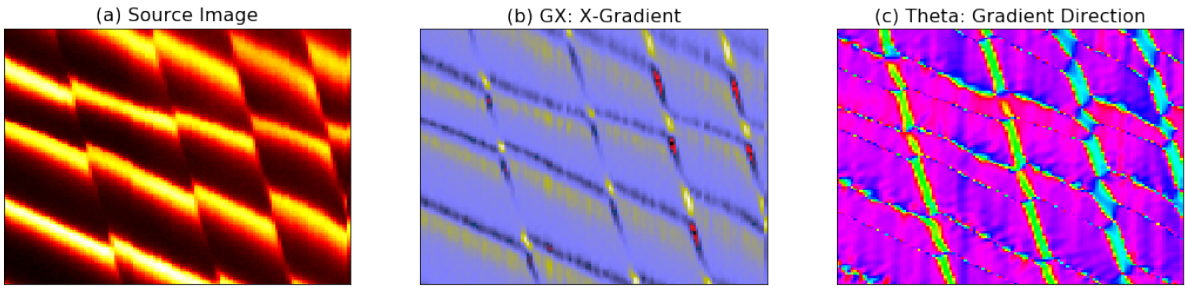


Figure 3.3: Sobel filtering applied to a charge stability diagram. Image generated in Python.

The first step in identifying transitions in a charge stability diagram is to transform it into an image that highlights pixels on a transition. It was found that the 2-dimensional gradient direction at transition pixels differed from the rest of the image. Figure 3.3(c) visualises this; the direction of the gradient is mostly uniform over the charge stability diagram, yet it changes for pixels on a transition. This can be attributed to the spike in horizontal gradient as seen in Figure 3.3(b) caused by the shifting of coulomb peaks. Hence, the solution needs to compute the 2-dimensional gradient direction for each pixel. A function was written to compute this; `calculate_theta_matrix()`. The first line of `find_transitions()` calls this function:

```
theta = calculate_theta_matrix(Z, filter=True)
```

`Z` is the charge stability diagram matrix, and `filter` is an optional filtering applied to remove noise. Firstly, an optional binomial filter window is applied to the `Z` matrix:

$$Z = \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} * Z$$

The x and y gradient matrices are then calculated by computing a 2-dimensional convolution of the image with Sobel gradient operators G_x and G_y [17]. The same filter window as above is

optionally applied to the resulting matrices:

$$G_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * Z \quad \text{and} \quad G_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * Z$$

The x and y gradient matrices can then be used to calculate the gradient direction `theta` matrix:

$$\Theta = \arctan(\frac{G_y}{G_x})$$

Figure 3.3(b) is an example of a G_x matrix, and Figure 3.3(c) is an example of a `theta` matrix.

3.2.2 Filtering Theta

Once `theta` has been calculated, the solution uses this to determine which pixels lie on transitions. As seen in Figure 3.3 the gradient direction on transitions will differ greatly from anywhere else in the image. Thus, to determine which pixels lie on a transition, the solution compares the most common value in `theta` to each pixel. The most common value in a set of numbers is the mode, hence the function `find_matrix_mode()` was written to find the most common theta value.

```
theta_mode = find_matrix_mode(theta)
```

The difference is then calculated using the equation:

```
theta_dif = 1 - np.cos(theta_mode -theta)**2 #
```
$\Theta_{dif} = cos(\Theta_{mode} - \Theta)^2$

This equation is used because it is only sensitive to significant variances in theta, which works well for the solution.

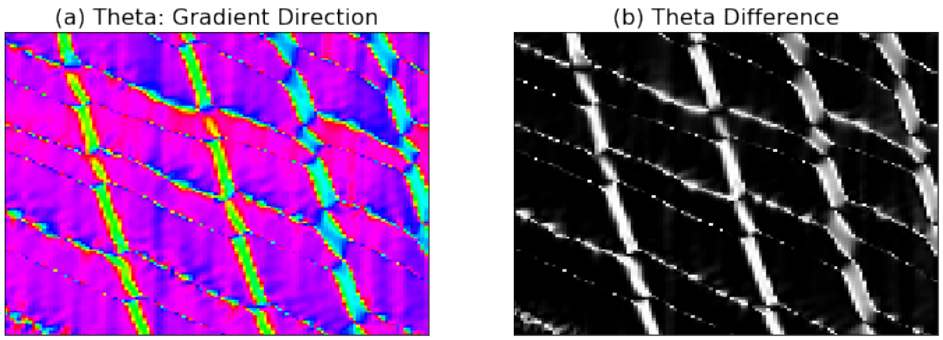


Figure 3.4: Theta vs theta difference. Image generated in Python.

Pixels that lie on transitions are then clearly highlighted in the resulting `theta_dif` matrix. Figure 3.4(b) depicts this, with lighter pixels representing those more likely to lie on a transition.

3.2.3 Hough Transform

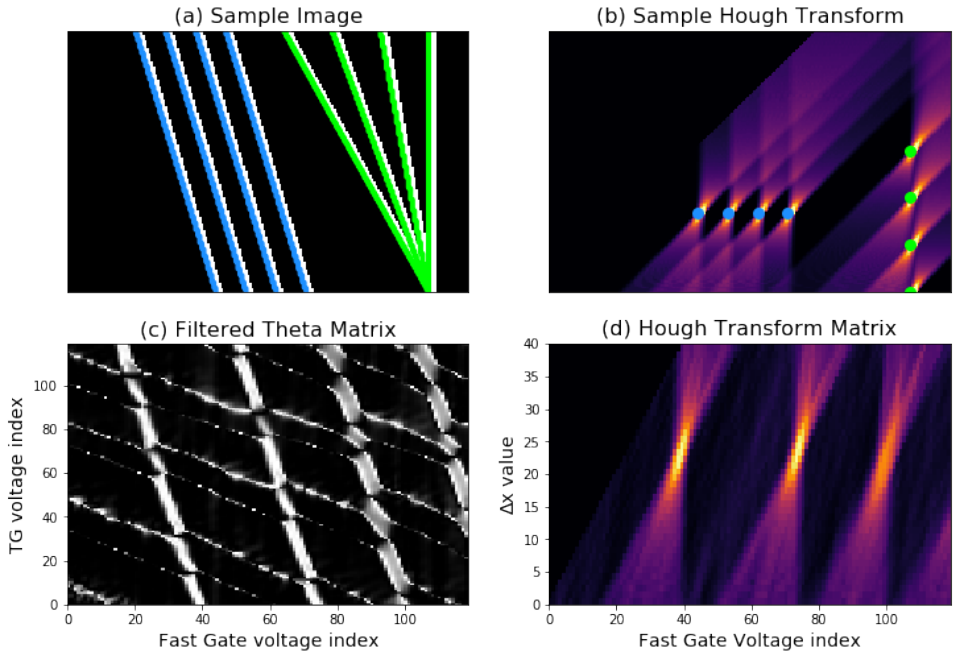


Figure 3.5: Source images vs Hough transforms. Image generated in Python.

Now that the solution is able to identify pixels likely to lie on a transition, it must be able to find lines in the `theta_dif` matrix corresponding to transitions. To do this, a function `calculate_hough_transform()` has been written to implement a modified version of the Hough transform. Essentially, this scans every possible line defined by a set of gradients and

start points. For each line, the sum of intersecting pixels is stored into a matrix at a point corresponding to its gradient and position.

This is best understood with Figure 3.5(a) and Figure 3.5(b) which depict a number of lines with constant gradients and different start points in blue, along with a number of lines with different gradients and constant start points in green. This is to outline how the Hough transform plots the lines in terms of start point and gradient. Figure 3.5(c) and Figure 3.5(d) showcases how this is applied to a real charge stability diagram. This implementation differs from a regular hough transform in a number of ways:

- Scanned variables : Instead of using the Hough transform convention of scanning for lines in the form $r = x \cdot cos\theta + y \cdot sin\theta$, this implementation scans for lines in the form $x = \frac{y}{m} - b$. The chosen form is simply the traditional $y = m \cdot x + b$ rearranged to scan for the x-intercept instead of y. This form was chosen for two main reasons:

  - Firstly, the gradient of lines is of importance, as it will reveal the relative coupling of gates to donors. The line's gradient can be easily read from the chosen form; it is simply the m value. As seen in Figure 3.5 the gradient in the image can be easily mapped to the Hough space.

  - Secondly, transitions do not always intercept the y-axis of a charge stability diagram, thus using a y-intercept is not useful when looking at a scan. Rearranging to scan for the x-intercept means that the Hough space and source charge stability diagram both share the same x-axis, which is favourable for ease of viewing, and makes the code more intuitive.

- Gradient scan range : At this stage, code efficiency is substantially important. If this function is to be called many times when scanning a set of many charge stability diagrams, it should only compute what is necessary. To improve efficiency, only a certain range of gradients are scanned. This can be done since it is known that transitions cannot have a positive gradient and cannot have a gradient of -1 (this would imply the donor is more coupled to the top gate than to the donor gates).

After computing the Hough transform, a 3x3 averaging window is convolved with the result to smooth the raw hough transform.

$$\texttt{hough\_filt} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} * \texttt{hough\_raw}$$

3.2.4 Peak Detection

Due to the way this section is completed, detecting peaks is as simple as finding the index of the maximum value in the Hough transform. If the peak value is not high enough, no transition is found. If the peak value is high enough, it is considered to be valid. Details about the transition are compiled into a dictionary, then added to a transition list as described in Section 3.2.6.

After a peak has been found, it is removed by deleting it's pixels from the `theta_dif` matrix and re-calculating the Hough transform for the modified pixels. The process of finding the maximum value in the Hough transform and deleting the peaks is repeated until no more substantial peaks remain. This is visualised in Figure 3.6.

There are two main reasons peak detection was done this way as opposed to conventional peak detection methods:

- Simplicity : Traditional peak detection methods have many variables to fine-tune what is defined as a peak. Especially in 2-dimensions, not only are these complicated, but can be innaccurate, finding false peaks. By incrementally deleting peaks, the solution ensures that only the most prominent peaks corresponding to transitions are found.

- Handling overlaps : When transitions are nearby, the Hough transform lines can pass through both transitions, creating a false peak in the Hough matrix. By incrementally deleting peaks, any line that would cross two transitions only crosses one, because the other has been removed. This is seen in Figure 3.6 where the first scan shows four peaks. However, after incrementally deleting transition pixels, it becomes apparent that there are only three transitions.

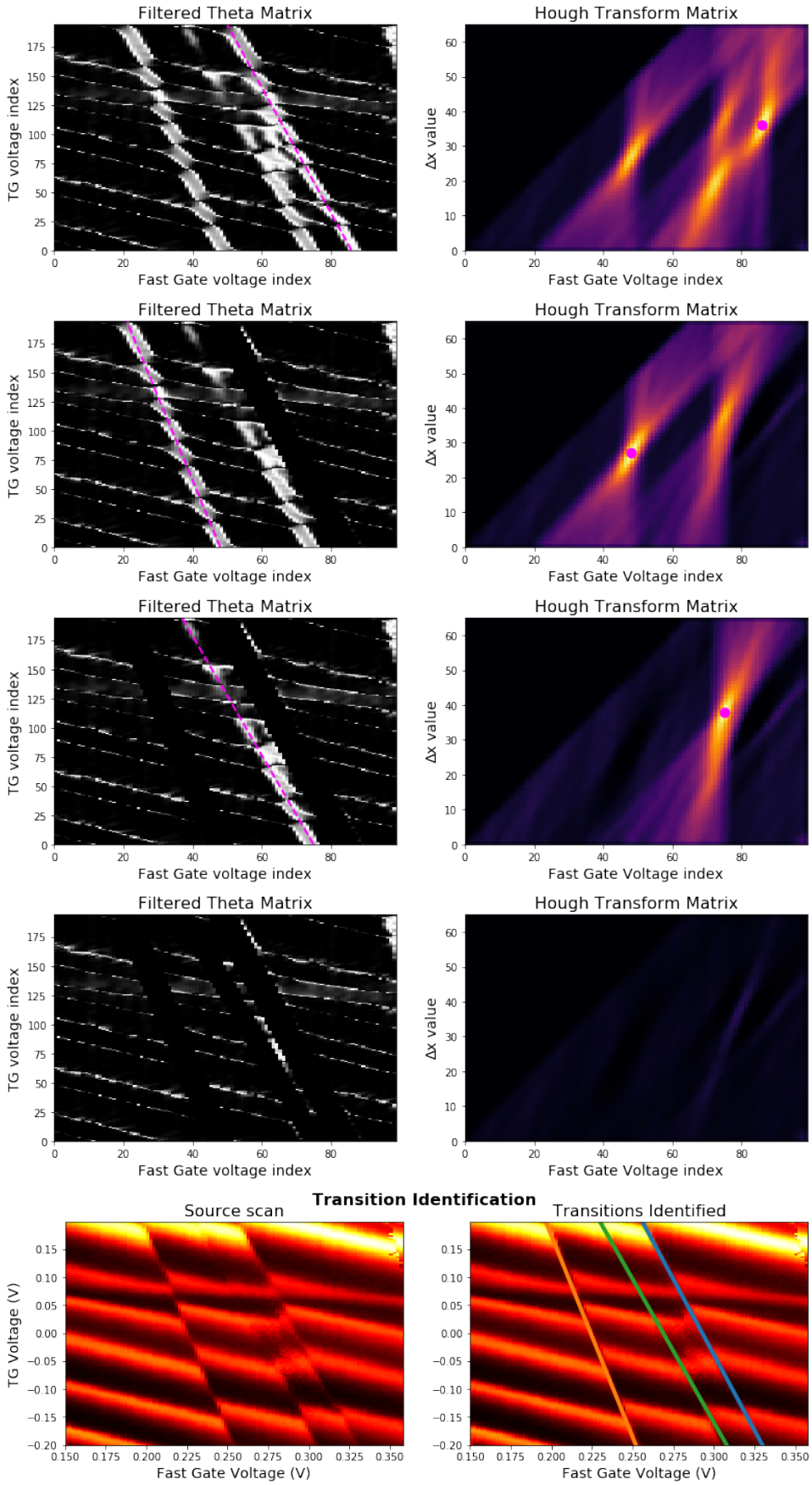


Figure 3.6: Incrementally deleting transition pixels to find peaks. Image generated in Python.

3.2.5 Charge Transfer

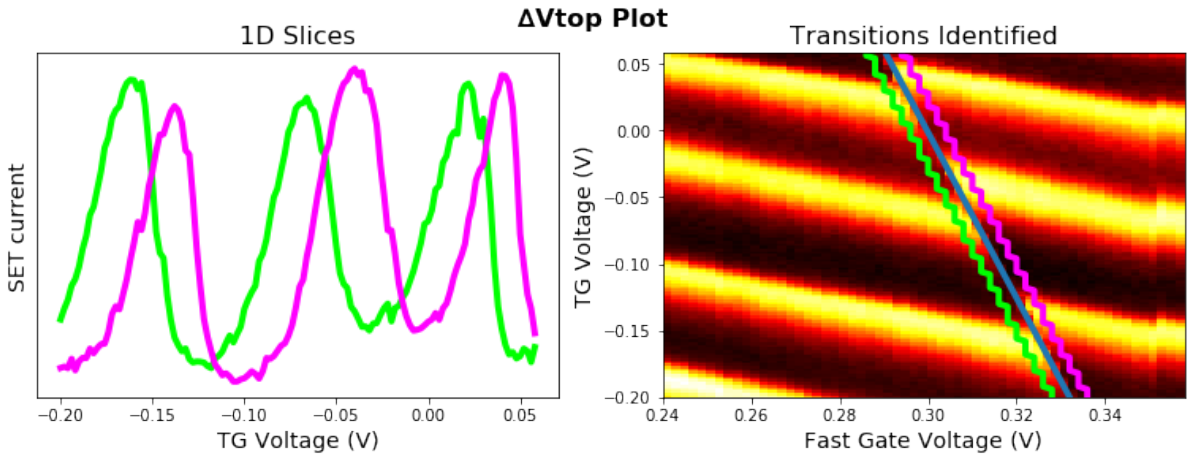


Figure 3.7: Coulomb peak shift due to a charge transfer event. Image generated in Python.

As an additional capability, the coulomb pea voltage shift ΔV_{top} is calculated for each transition. This is the how much voltage the coulomb peaks are shift by at a transition. ΔV_{top} can be used to derive Δq, which is the charge induced on the SET island due to a charge transfer event [5]:

$$\Delta V_{top} = \frac{\Delta q}{C_{top}}$$

where C_{top} is the capacitance between SET island and top gate

This is done by taking two slices from either side of a transition then measuring the distance between peaks. Figure 3.7 visualises the slices taken either side of the transition.

3.2.6 Finding Transitions in 2-Dimensions

Using the methods described in the sections above, `find_transitions()` is able to take a charge stability diagram and return a list of all transitions found. The sample case from Section 3.1.1 will be revisited in more detail:

Sample case:

```
from transitions import*
# DB = DBL_DBR gate voltage array
# TGAC = TGAC voltage array
# DC_voltage  = Charge stability diagram with axes [DB,TGAC]
transitions = find_transitions(DB, TGAC, DC_voltage,
                               true_units=True,
                               plot=True)
display(transitions)
```

Sample output:

```
[{'dVtop': 0.016000000000000014,
  'gradient': -7.0000000000000488,
  'gradient_error': 5.9027777777777786,
  'intensity': 0.8181497829646055,
  'location': 0.17399999999999999},
 {'dVtop': 0.012000000000000011,
  'gradient': -5.1739130434782972,
  'gradient_error': 4.356060606060602,
  'intensity': 0.77731879466478193,
  'location': 0.33400000000000002},
 {'dVtop': 0.012000000000000011,
  'gradient': -6.611111111111156,
  'gradient_error': 5.5727554179566594,
  'intensity': 0.56845723964873407,
  'location': 0.29799999999999999}]
```

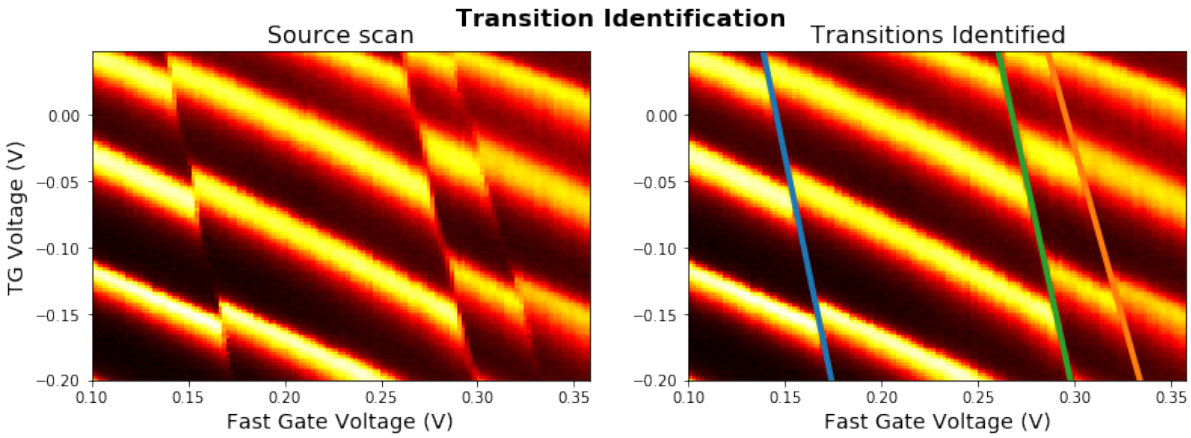


Figure 3.8: Sample transition identification. Image generated in Python.

`find_transitions()` takes a charge stability diagram as input (`DC_voltage`), along with arrays of the gate voltages on the x and y axis (`DB` and `TGAC`). An optional argument `plot` can be set that will automatically plot the source charge stability diagram next to duplicate image with the identified images overlaid. `true_units` is an optional argument to either return the results in voltages, or in terms of array indices. This is mainly used in debugging and internally within the solution. By default this is set to return all results as voltages. `plot` is an optional plotting feature. There are three options:

- `bool : False` : No results are plotted
- `bool : True` : Plots the source charge stability diagram next to a duplicate with the identified transitions highlighted. This is can be done externally by calling the function `plot_transitions()`.
- `str : 'Complex'` : Plots the Hough transform (`hough_filt`) and `theta_dif` matrices for each transition. This is helpful to visualise the process of incrementally deleting transitions, such as in Figure 3.6.

Each transition is characterised in the form $x = \frac{y}{m} - b$, where y is top gate voltage, x is donor gate voltage, m is the gradient, and b is the donor gate intercept. The function's output is a list of transition dictionaries; each dictionary corresponds to a transition and contains the following properties:

- `gradient` : The gradient of the transition in the form $\frac{\Delta V_{TG}}{\Delta V_{DG}}$; the m component of the characteristic equation.
- `gradient_error` : Due to the discretised nature of images, pixels can only be in certain places. Thus there is a degree of uncertainty to the gradient.
- `intensity` : A measurement of how certain the algorithm is that the identified transition is a real transition. This is found as the peak's value in the Hough transform.
- `location` : The voltage where the transition intersects the bottom of the charge stability diagram. This is not the b value, rather the intercept of the charge stability diagram donor

gate axis. This value is returned rather than the b value, as it is more useful when tracking donors, and is easier to visualise.

- `dVtop` : This is ΔV_{top}, how much voltage the coulomb peaks shift by at the transition.

`find_transitions()` is very effective at automatically identifying transitions within a charge stability diagram. It does this effectively and quickly (case above took 209ms). Therefore this function successfully addresses one of the main goals of this thesis. The data able to be gathered about transitions is not only useful to researchers, but is a valuable step towards the next stage of this thesis; donor tracking.

3.3 Tracking Transitions

3.3.1 Finding Transitions in 3-Dimensions

Multi-dimensional charge stability diagrams are generated by taking a series of fast 2-dimensional scans. A donor gate and the top gate can be driven by a fast AC source, allowing them to be scanned very quickly, while a third gate can be scanned more slowly. This creates a series of scans in the fast gate and top gate plane while the slow gate is varied. Thus when plotting, the names 'fast' and 'slow' are used to represent the gates. When finding the capacitive coupling for many gates, it is significantly faster to take many 3D scans varying one slow gate at a time, rather than taking higher dimensional scans.

A function `find_transitions_3D()` has been written, that calls the function `find_transitions()` on slices of a 3-dimensional charge stability diagram. This function locates transitions in each 2-dimensional slice, then later a tracking algorithm can be called to correlate the 2D transitions. Below is an example of this function:

Sample Case:

```
# slow  = DFR gate voltage array
# fast  = DBL_DBR gate voltage array
# TG    = TGAC voltage array
# DC_voltage  = Charge stability diagram with axes [DFR,TGAC,DB]
```

```
transition_list = find_transitions_3D(slow,fast,TG,DC_voltage);
plot_transitions_3D(slow, fast, TG, DC_voltage, transition_list, slices=True)
```

Sample output:

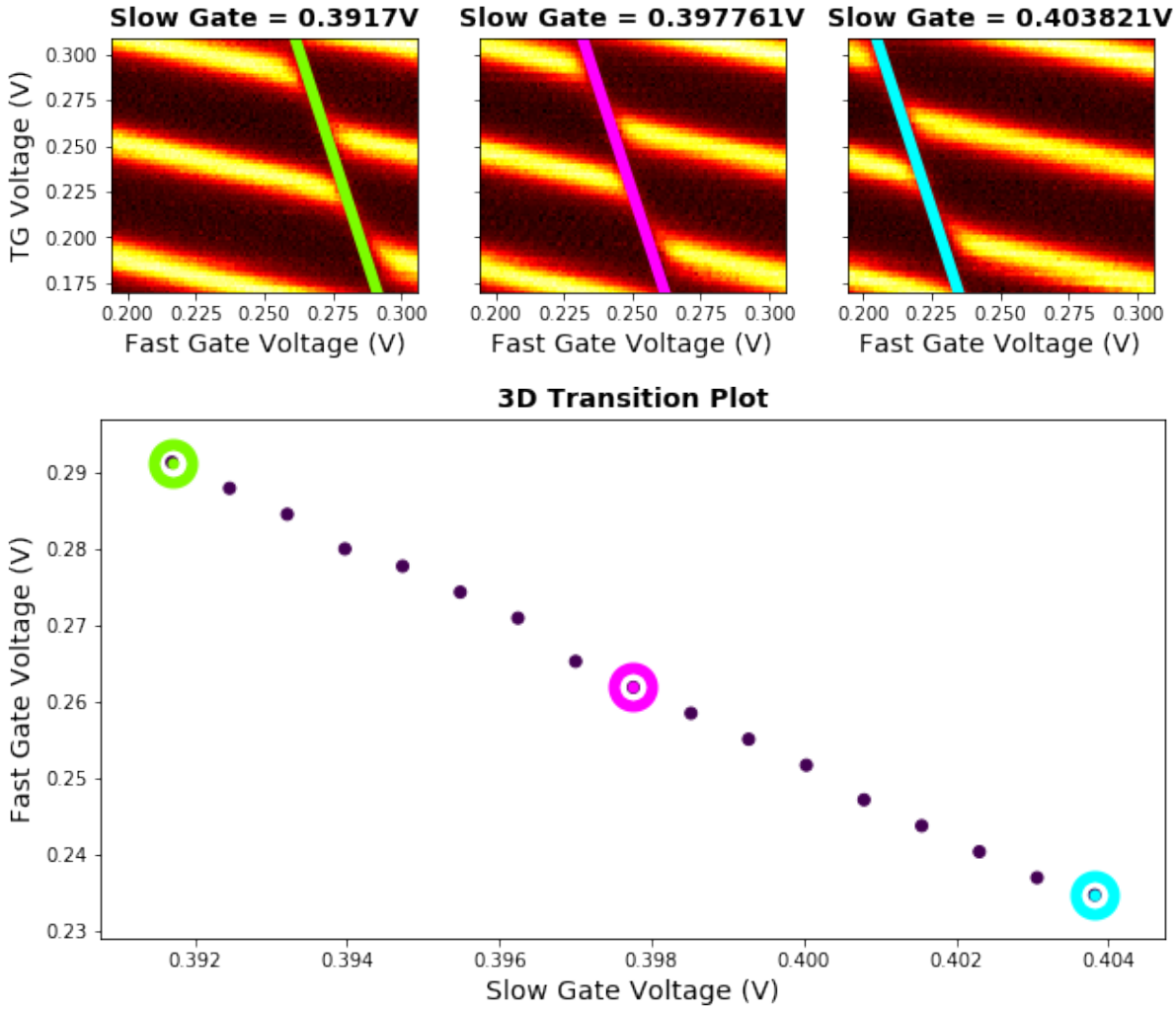


Figure 3.9: Sample 3D transition tracking. Image generated in Python.

`find_transitions_3D()` takes the same arguments as the 2D version, though now the charge stability diagram `DC_voltage` has 3-dimensions, and three gate voltage arrays are provided: `DB`, `DFR`, and `TGAC`. The function's output is a list of transition lists for each 2D slice (`trans_list` in the example). `plot_transitions_3D()` has been called with the optional argument `slices = True` which enables plotting of 2D slices along the slow gate axis. For each point on the slow gate axis, `find_transitions()` is called on a 2D slice in the fast gate and TG plane. The 2D transition locations are plotted on the fast gate axis, and 2D gradients represented as colour, with the colour bar as reference. After scanning each value on the slow

gate axis, the transition's path can be seen, and a tracking method can be applied.

3.3.2 Tracking Single Transitions in 3-Dimensions

`track_transitions_single()` is a function that has been written to correlate transitions found using `find_transitions_3D()`. This function is only capable of tracking a single transition, however it is accurate and fast. It does this by fitting a line $y = m \cdot x + b$ to the identified transition points using least squares approximation. Below is an example:

Sample Case:

```
# slow  = DFR gate voltage array
# fast   = DBL_DBR gate voltage array
# TG = TGAC voltage array
# DC_voltage  = Charge stability diagram with axes [slow,TG,fast]
trans_list = find_transitions_3D(slow,fast,TG,DC_voltage)
tracked = track_transitions_single(slow, fast, TG, DC_voltage, trans_list)
plot_transitions_3D(slow, fast, TG, DC_voltage, trans_list, tracked)
display(tracked)
```

Sample output:

```
[{'TG/fast gradient': -4.6285897182597706,
  'TG/slow gradient': 0.96460389629642296,
  'fast intercept': 2.1711063596862799,
  'fast/slow gradient': -4.7984356439271565}]
```

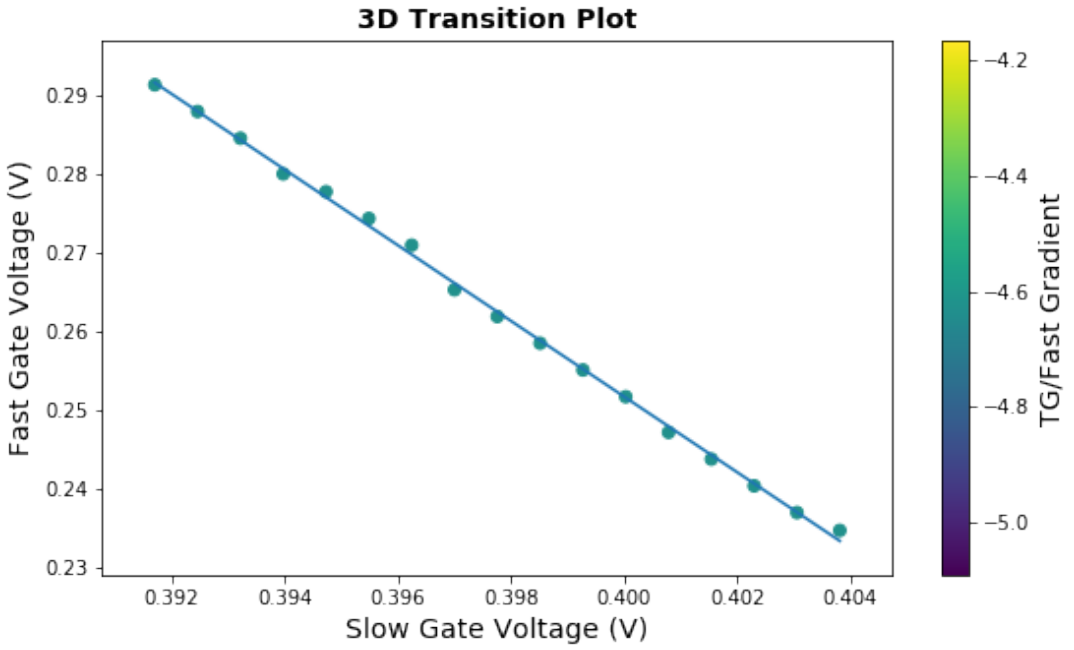


Figure 3.10: Automated tracking of a single transition. Image generated in Python.

This function returns a dictionary of relevant values:

- `TG/fast gradient` : The mean gradient of the transitions in the 2D slices. Can be seen in the colour bar next to the plot. This represents the relative capacitive coupling of top and fast gates to the donor:

$$\frac{\Delta V_{TG}}{\Delta V_{fast}}$$

- `fast/slow gradient` : The gradient of how the transitions move in the fast axis with respect to the slow axis, seen as the slope of the plot. This is the m value in the fit line and represents the relative capacitive coupling of the fast and slow gates to the donor:

$$\frac{\Delta V_{fast}}{\Delta V_{slow}}$$

- `TG/slow gradient` : This represents the relative capacitive coupling of the top and slow gates to the donor. This is implicitly calculated from:

$$\frac{\Delta V_{TG}}{\Delta V_{slow}} = \frac{\Delta V_{TG}}{\Delta V_{fast}} \cdot \frac{\Delta V_{fast}}{\Delta V_{slow}}$$

- `fast intercept` : The voltage where the tracked transition would intersect the fast gate axis. This is the b value in the fit line.

The values from this function may then conveniently be read into a capacitance matrix, which is a table containing the relative capacitive coupling of the gates to a specific donor:

Gate 1	**Gate 2**	$\frac{\Delta V_1}{\Delta V_2}$ **Gradient**
`TGAC`	`DBL_DBR`	-4.6285
`DBL_DBR`	`DFR`	-4.7984
`TGAC`	`DFR`	0.9646

`DBL_DBR` = fast, `DFR` = slow, `TGAC` = TG

As seen in Figure 3.10, the solution can automatically track transitions over a third dimension and determine the relative capacitive coupling of three gates to the tracked donor. All this

is done in one line of code and computes in approximately one second ($1.03s$). This achieves a goal of this thesis to automatically track a transition and calculate the relative capacitive coupling of a transition to each of the scanned gates.

While `track_transitions_single()` works very well for single transitions, it does not work if multiple transitions are present. This is because the least-squares approximation is naively used, assuming that all transitions are part of the same line. To track multiple transitions at once, a better method is needed.

3.3.3 Tracking Multiple Transitions in 3-Dimensions

`track_transitions_multi()` attempts to address the shortcomings of `track_transitions_single()`. It does so by calculating the least squares approximation of every possible combination of 2D transition points and finding the lines which fit most accurately. It returns these tracking lines as a list of dictionaries with the same elements as in `track_transitions_single()`.

The function is incredibly inefficient and troublesome to work with, though it does produce promising results for small datasets. It is highly advised to not use this function, as it has used over 16GB of RAM in the test computer for large scans.

First, a set of almost all possible combinations of transitions are generated. The combinations must have at least two thirds as many transitions as there are 2D scans. The number of combinations is proportional to:

$$N_t^{N_s} \cdot (N_s - 1)!$$

Where N_t is number of transitions per 2D scan, and N_s is number of scans.

Next, the least squares line approximation is used for each of these combinations. The point error for each tracking line is calculated, as well as the variance in 2D gradient for each transition on the line. The tracking lines are then sorted based on combined error. A list of unique tracking lines are returned in order of best to worst fit. Below is an example of this function:

Sample Case:

```
# DB   = DBL_DBR gate voltage array
# DFR  = DFR gate voltage array
# TGAC = TGAC voltage array
# DC_voltage  = Charge stability diagram with axes [DB,TGAC,DFR]
```

```
trans1 = find_transitions_3D(DFR,DB,TGAC,Z1)
fit1 = track_transitions_multi(DFR,DB,TGAC,Z1,trans1)
plot_transitions_3D(DFR,DB,TGAC,Z1,trans1,fit1)
```

Sample output:

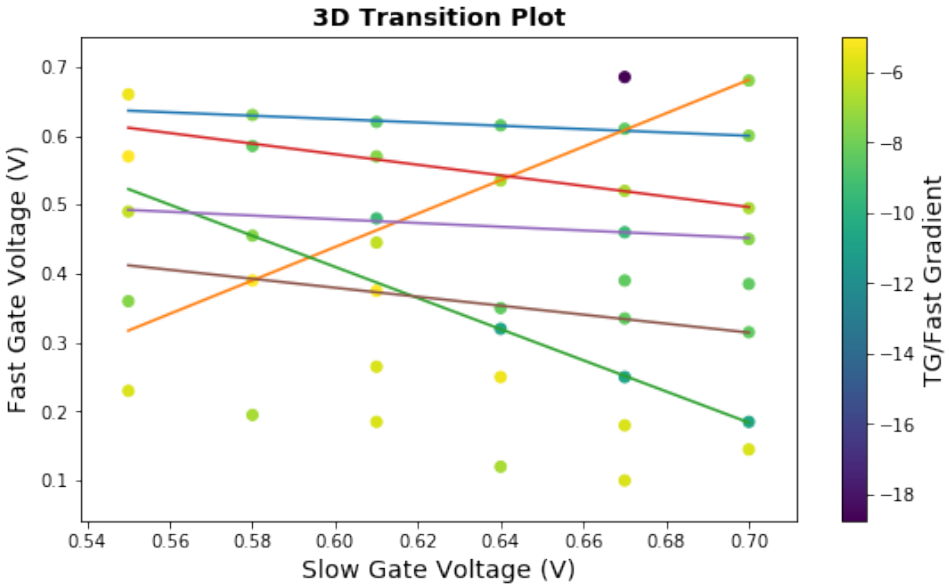


Figure 3.11: Unreliable tracking of multiple transitions while varying DFR. Image generated in Python.

While this is not 100% accurate, there are some tracking lines which do seem to track the transitions very closely. In addition, these transitions have the same gradient, which indicates they are the same one. Since these two tracking lines fit the transitions reasonably well, the output can be collected to two capacitance matrices. Below is the tracking lines and capacitance matrices for the two donors.

Case 1:

```
# DB   = DBL_DBR gate voltage array
# DFR  = DFR gate voltage array
# TGAC = TGAC voltage array
# DC_voltage  = Charge stability diagram with axes [DB,TGAC,DFR]
trans1 = find_transitions_3D(DFR,DB,TGAC,Z1)
fit1 = track_transitions_multi(DFR,DB,TGAC,Z1,trans1)
plot_transitions_3D(DFR,DB,TGAC,Z1,trans1,fit1)
display(fit1)
```

Sample output:

```
[{'TG/fast gradient': -7.9166666666666732,
```

```
  'TG/slow gradient': 32.598039215686285,
  'fast intercept': 0.76985714285714313,
  'fast/slow gradient': -0.24285714285714297},
 {'TG/fast gradient': -7.3674242424242484,
  'TG/slow gradient': 9.6096837944664344,
  'fast intercept': 1.0331666666666655,
  'fast/slow gradient': -0.76666666666666483}]
```

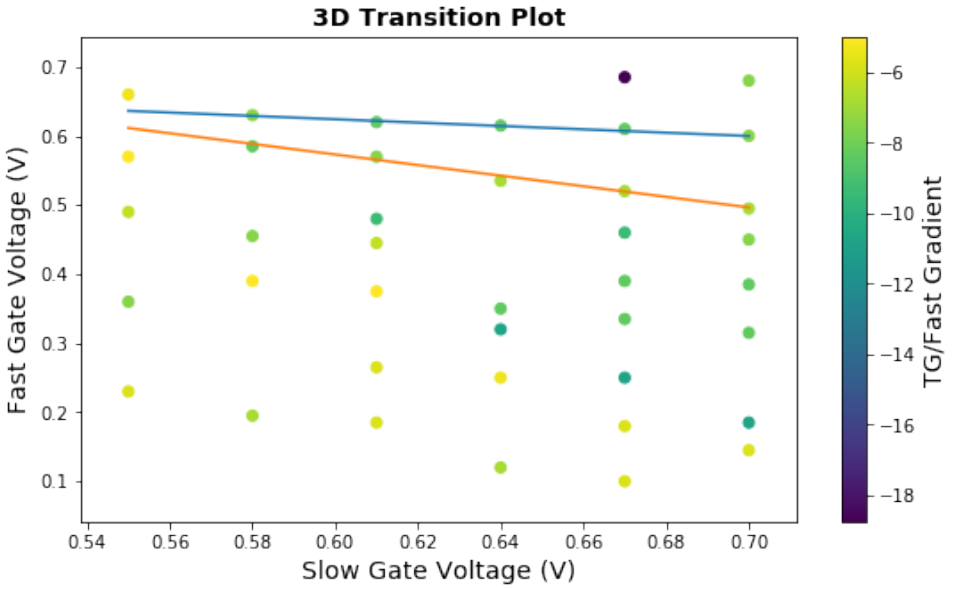


Figure 3.12: Automated tracking of multiple transitions while varying DFR. Image generated in Python.

Donor 1 Capacitance Matrix

Gate 1	Gate 2	$\frac{\Delta V_1}{\Delta V_2}$ Gradient
TGAC	DBL_DBR	−7.9166
DBL_DBR	DFR	−0.2428
TGAC	DFR	32.5980

Donor 2 Capacitance Matrix

Gate 1	Gate 2	$\frac{\Delta V_1}{\Delta V_2}$ Gradient
TGAC	DBL_DBR	−7.3674
DBL_DBR	DFR	−0.7666
TGAC	DFR	9.6096

While the case above tracks donors using `DFR` (Donor Front Right gate) as the slow gate, the same can be done with `DFL` (Donor Front Left gate):

Case 2:

```
# DB   = DBL_DBR gate voltage array
# DFL  = DFL gate voltage array
# TGAC = TGAC voltage array
# DC_voltage  = Charge stability diagram with axes [DB,TGAC,DFL]
trans2 = find_transitions_3D(DFL,DB,TGAC,Z2)
fit2 = track_transitions_multi(DFL,DB,TGAC,Z2,trans2)
```

```
plot_transitions_3D(DFL,DB,TGAC,Z2,trans2,fit2)
display(fit2)
```

Sample output:

```
[{'TG/fast gradient': -8.3414502164502249,
  'TG/slow gradient': -145.975378787876,
  'fast intercept': 0.56799999999999939,
  'fast/slow gradient': 0.057142857142858293},
 {'TG/fast gradient': -8.601866883116891,
  'TG/slow gradient': 7.7694281524926643,
  'fast intercept': 1.2635714285714297,
  'fast/slow gradient': -1.1071428571428588}]
```

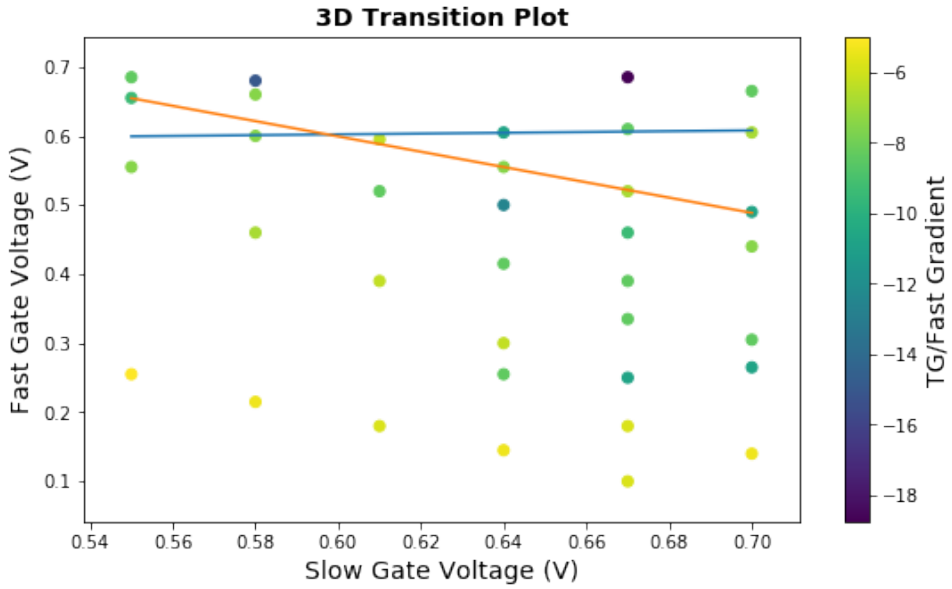


Figure 3.13: Automated tracking of multiple transitions while varying DFL. Image generated in Python.

These two 3D scans overlap when both `DFR` and `DFL` are at $0.67V$. Thus it can be determined that the two tracking lines in each scan are tracking the same two donors; the blue and orange lines across both `DFR` and `DFL`. Thus, the tracking results can be collected into one capacitance matrix for each donor:

Donor 1 Capacitance Matrix

Gate 1	Gate 2	$\frac{\Delta V_1}{\Delta V_2}$ Gradient
TGAC	DBL_DBR	−7.9166
DBL_DBR	DFR	−0.2428
TGAC	DFR	32.5980
DBL_DBR	DFL	0.05714
TGAC	DFL	−145.9753

Donor 2 Capacitance Matrix

Gate 1	Gate 2	$\frac{\Delta V_1}{\Delta V_2}$ Gradient
TGAC	DBL_DBR	−7.3674
DBL_DBR	DFR	−0.7666
TGAC	DFR	9.6096
DBL_DBR	DFL	−1.1071
TGAC	DFL	w 7.7694

Further work is required for this function to flawlessly track multiple transitions and deal with larger datasets. However, `track_transitions_multi()` has produced some promising results, and is able to automatically estimate tracking lines for small datasets with multiple transitions.

Chapter 4

Results

4.1 Testing

For the duration of testing, eight charge stability diagrams with 0 to 7 transitions were used as test subjects. Tracking algorithms were not officially benchmarked, because their correct operation depends entirely on `find_transitions()`. Thus if `find_transitions()` work correctly, then tracking algorithms will work correctly, apart from any known issues.

4.1.1 Decimation

Decimation testing will check to see how the `find_transitions()` can handle down-sampling. A decimation factor d, means the source charge stability diagram will only be sampled every d points. The following code was run as a test. `d` was increased until `find_transitions()` started showing errors for each test case. An error is considered to be a missed transition, or a false identified transition.

```
d = 2 #decimation factor
DB_test = DB[::d]
TGAC_test = TGAC[::d]
DC_voltage_test = DC_voltage[::d,::d]
find_transitions(DB,TGAC,DC_voltage,plot=True)
find_transitions(DB_test,TGAC_test,DC_voltage_test,plot=True);
```

Sample Output:

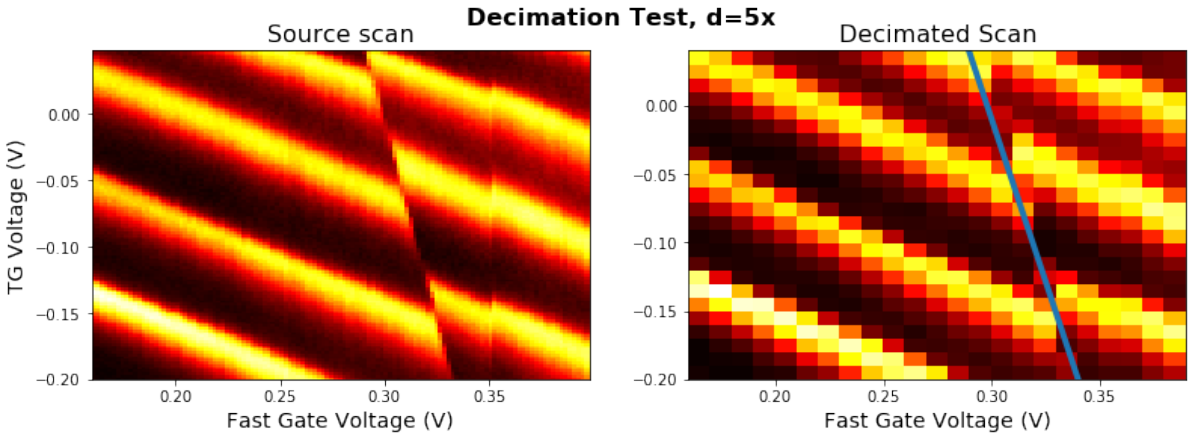


Figure 4.1: Decimation testing with d=5. Image generated in Python.

For the test cases, the following decimation factors were tolerated with no error:

Average = 2.2

Minimum = 1 (no decimation)

Maximum = 5

The results indicate `find_transitions()` on average has no errors up till 2.2x decimation. This is neither exceptionally good nor bad, though it means the function works better with finer voltage step sizes. It is worth noting that test cases usually had multiple transitions in each diagram. Decimation is tolerated better when fewer transitions are present.

Full test data is included in Appendix A - Test Results.

4.1.2 Sample Size

Sample size testing checks to see how small of a window can be taken in order for `find_transitions()` to still be able to identify a transition. For this case, charge stability diagrams were zoomed in until the function stopped being able to identify transitions. The following code was run as a test.

```
DB_test = DB[160:180]

TGAC_test = TGAC[200:250]
```

```
DC_voltage_test = DC_voltage[200:250,160:180]
find_transitions(DB_test,TGAC_test,DC_voltage_test,plot=True);
```

Sample Output:

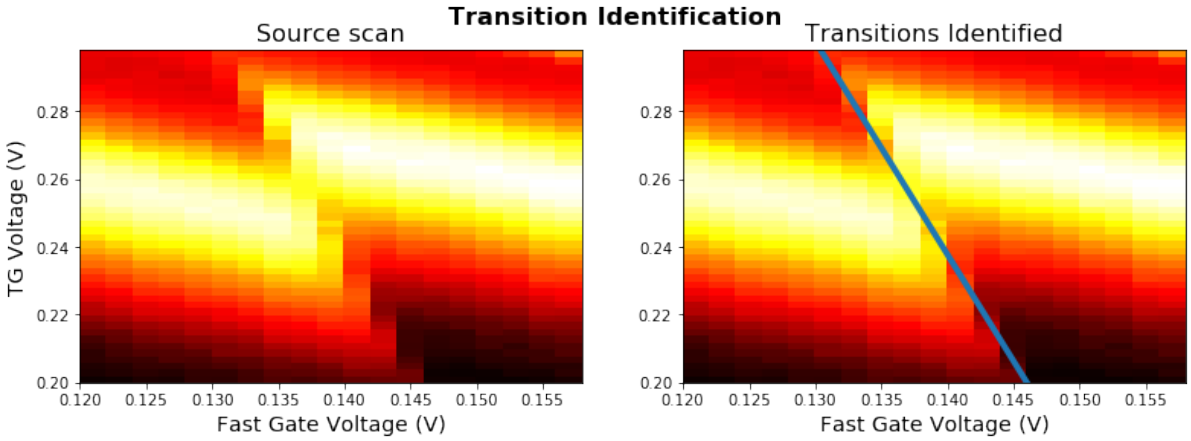


Figure 4.2: Sample size testing, with size 20x50. Image generated in Python.

Generally, there is no issue with identifying a transition in a 20x50 pixel charge stability diagram. The smallest window successfully tested, was a 20x25 window. This is very good, since scans are almost always larger than this.

4.1.3 Noise

Noise testing checks how tolerant `find_transitions()` is to noise. A random uniform noise was applied. The following code was run as a test. `alpha` represents the magnitude of the unifrom noise, and was increased until `find_transitions()` started showing errors for each test case. An error is considered to be when a transition is missed, or when a false transition is identified.

```
alpha = 0.11
DB_test = DB
TGAC_test = TGAC
DC_voltage_test  = DC_voltage + alpha*np.random.uniform(-1,1,size=DC_voltage.shape)
find_transitions(DB,TGAC,DC_voltage,plot=True)
find_transitions(DB_test,TGAC_test,DC_voltage_test,plot=True);
```

Sample Output:

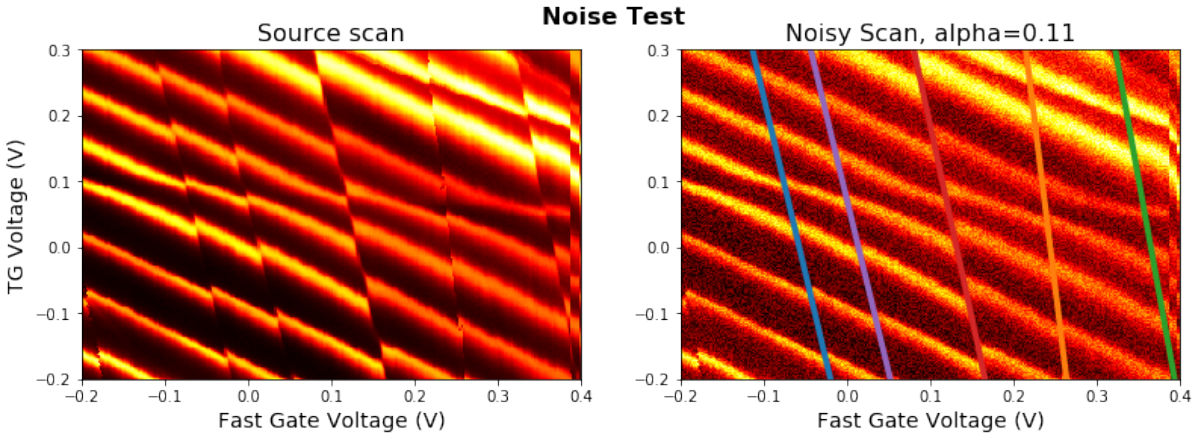


Figure 4.3: Noise testing with alpha=0.11. Image generated in Python.

For the test cases, the following `alpha`s were tolerated with no error:

Average = 0.09

Minimum = 0.05)

Maximum = 1.1

The results mean that `find_transitions()` on average has no errors up till an alpha of 0.09, which relates to a maximum variance of approximately 11%. This is quite good, however it is not necessarily indicative of how the algorithm handles measurement noise. Measurement noise does not only affect the SET current, it can affect the electrochemical potential of the island and donor.

Full test data is included in Appendix A - Test Results.

4.1.4 Speed

Speed testing will reveal how fast `find_transitions()` is able to identify transitions. This is important because any tracking method will be calling `find_transitions()` many times, and thus is must be quick. Tests were done on 24 different charge stability diagrams, of varying size and number of transitions:

```
# DB_test = DB
```

```
# TGAC_test = TGAC
# DC_voltage_test = DC_voltage
%time find_transitions(DB_test,TGAC_test,DC_voltage_test);
```

Full results are included in APPENDIX REFERENCE. The results showed that the main factor is charge stability diagram size. While the number of transitions did influence the result, it was not significant. The following are average times for different sized scans with between 3 and 7 transitions:

```
300x250:  1.31s
150x250:  653ms
150x125:  244ms
```

The time taken to identify transitions is roughly proportional to the number of pixels in the source charge stability diagram. Additionally, large scans were more likely to have more transitions, which had a small but noticeable impact. A least squares approximation was used to determine a decently accurate equation for how long a scan will take:

$$\text{Time}_{ms} = N_t \cdot 10 + N_p \cdot 0.0068 + N_t \cdot N_p \cdot 0.0017$$

Where N_t is number of transitions, and $N_p = len(x) \cdot len(y)$ is number of pixels

A nominal 100x100 scan with one transition will take approximately 73ms. Since the 2D scans are very fast, a 100x100x20 tracking scan takes approximately 1.16s, which is still very fast. This speed will pose no obstacle to researchers, and thus is very acceptable. Though much optimisation has already been done to make the code 9.7x faster than the first iteration, more optimisation is likely possible.

Full test data is included in Appendix A - Test Results.

4.1.5 Accuracy

Accuracy is simply to run `find_transitions()` on multiple test cases and determine how many identified transitions are actually errors. An error is considered to be when a transition

is missed, or when a false transition is identified. Of all eight test cases, 38 correct transitions were identified, and only one error was found. This is shown in Figure 4.4 where the transition was not correctly identified due to its presence being too faint. Ultimately this comes to a success ratio of:

$$\text{Success ratio} = \frac{38}{39} = 97.4\%$$

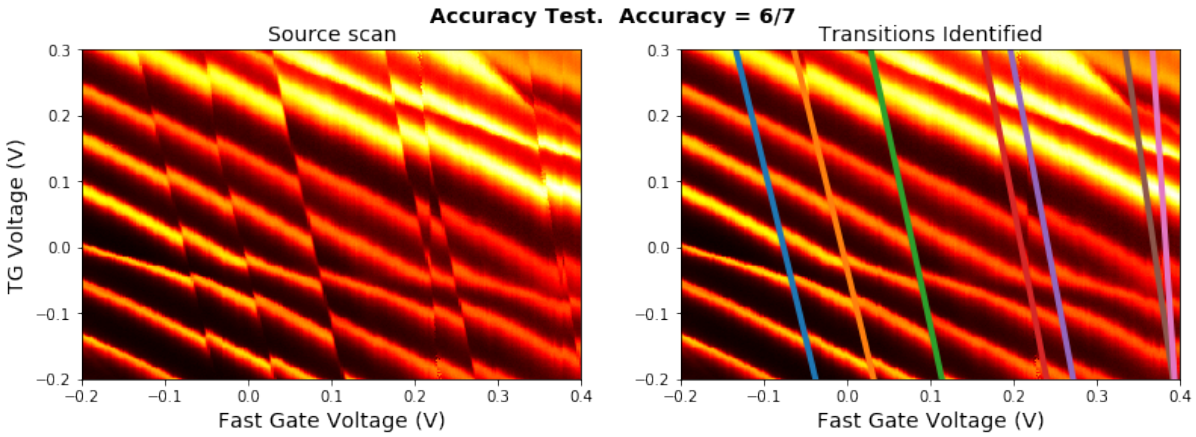


Figure 4.4: Accuracy error on right most transition. Image generated in Python.

This is very good, however it is advised that researchers visually validate the transitions using `plot_transitions()`. Visually verifying is very fast and merely involves glancing at highlighted transitions on a charge stability diagram. `find_transitions()`, `find_transitions_3D()`, and `track_transitions_single()` all have built in optional plotting for this purpose, along with the standalone functions `plot_transitions()` and `plot_transitions_3D()`.

Full test data is included in Appendix A - Test Results.

4.1.6 Testing Review

Due to the device refrigerators being unavailable for the majority of the development of this algorithm, testing had to be completed on saved data. This means only a limited number of test samples were available. However, the test results are promising:

- Average decimation factor = 2.2x : This means that most scans could be halved in resolution, and this solution would still be able to identify transitions. This is quite good.
- Nominal minimum sample size = 20x50 pixels : This is excellent. Most scans are much bigger than this, which means that they will almost always be big enough for transitions to be identified. If there is an error identifying a transition, it is most likely not this.
- Average noise handling up to $\pm 11\%$: This means the algorithm is quite good at handling noise in the SET current readout. However, noise will probably not only affect readout level, but influencing the donor and SET island electrochemical potential.
- Speed : A 100x100 2D scan with one transition takes 73ms. A 100x100x20 tracking scan takes approximately 1.6s. This is incredibly fast and will be no issue to researchers.
- Accuracy : Out of 39 identified transitions only one error was recorded. This is very good however it is advised that researchers visually validate the transitions where necessary.

4.2 Limitations

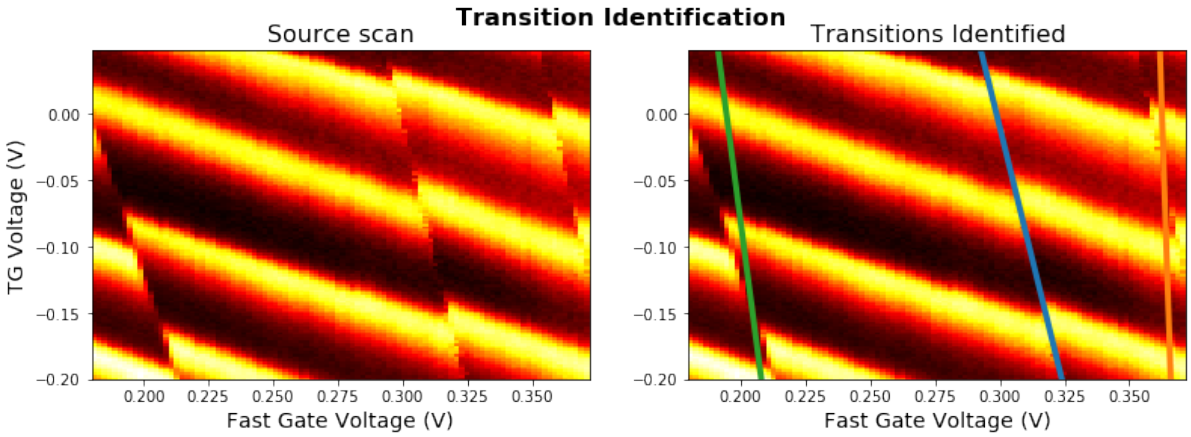


Figure 4.5: Limited accuracy when transitions cross the edge. Image generated in Python.

While the solution does work quite well for most cases, some limitations have been identified:

- One issue that has been identified, is the solution struggles to identify transitions that do not shift the coulomb peaks very much. This is shown in Figure 4.4 where the solution struggled to correctly identify a faint transition.
- Transitions at the edge of a charge stability diagram do not register properly. Figure 4.5 depicts how transitions that cross the edge can detected improperly.
- Tracking multiple transitions at once in a 3D scan is very inefficient and slow. This renders the function `track_transitions_multi()` unusable for large datasets. Section 3.3.3 discusses this issue in more depth.
- `track_transitions_multi()` Is not fully capable of confidently tracking multiple transitions at once, and may falsely identify a donor. Section 3.3.3 also discusses this in more depth.

4.3 Improvements

Apart from solving the limitations previously described, a number of improvements have been identified:

- Currently, the filtering kernels used to filter charge stability diagrams and Hough transforms are set values. It would be useful to have these filtering kernels as arguments for `find_transitions()`. Researchers would then be able to fine-tune the filtering kernel to get the best results.

- `track_transitions_multi()` somewhat works, although it cannot be scaled to a useable level and is not reliable enough. It is recommended that a completely different approach be used to implement this function in order for it to be useful.

- Live implementation. Currently, the solution is only capable of identifying transitions from saved scan data. It would be very useful for researchers to be able to identify transitions in real time, and be able to track how transitions move in real time when varying a third gate.

4.4 Solution Summary

`find_transitions()` has been shown to be capable of automatically identifying transitions within a charge stability diagram. It returns the location, gradient, and ΔV_{top} coulomb peak shift of each transition, along with confidence values for the location and gradient.

`find_transitions_3D()` and `track_transitions_single()` have been shown to be highly effective at automatically tracking the movement of a single transition while varying a third slow gate. Given a 3-dimensional charge stability diagram, the solution is able to automatically identify a transition's motion and return the relative capacitive coupling of the three gates to the donor.

`track_transitions_multi()` exhibited many issues, though also yielded some promising results. The unscalability and unreliability limit this function from being a full success. However, with selecting the right results, it was able to automatically generate the values of a capacitance matrix for two donors at once. This was then done again for a different gate, returning more values to add to the capacitance matrices.

Decimation, sample size, noise, speed, and accuracy tests have all determined that the solution is sufficiently adept in most cases where the charge stability diagram has been accurately gathered. The solution can be relied upon to automatically identify and track transitions with substantial accuracy, however it is still advised that researchers visually verify the identified transitions where possible. `plot_transitions()` and `plot_transitions_3D()` serve this purpose. They are accessible as standalone functions, can be called with the functions `find_transitions()`, `find_transitions_3D()`, and `track_transitions_single()` by setting the optional argument `plot = True`.

The solution is limited by it's inability to detect faint transitions, as well as transitions that cross the border of the charge stability diagram. Additionally `track_transitions_multi()` exhibited many issues with scalability and reliability. Improvements to this solution that will

provide extra functionality include: allowing for custom filter kernels, re-working `track_transitions_multi()`, and enabling live identification and tracking of transitions.

While the solution still has some limitations and improvements to be addressed it is highly effective in achieving the goals of this thesis: automated identification and tracking of transitions.

Chapter 5

Further Work

Work done in this thesis provides valuable development towards a number of future topics of research. These include:

- Donor Triangulation : A future extension of this thesis could pursue automatic donor triangulation which can estimate the physical location of donors. Automatically determining the capacitive coupling of gates to donors could one day be used to automatically locate the physical position of donors within the device. A lot of work is still required to map the relative capacitance coupling of gates to physical space, however the work done in this thesis is still highly valuable.

- Automated Tuning : It is possible to further automate this solution to find points along transitions that are most suitable for spin-dependent tunnelling. This would require more work locating tuning points and optimising readouts.

- Automated Compensation : If relative coupling capacitances are known for a donor, a system could be developed that would automatically compensate gate voltages while researchers manual adjust one gate.

- Donor Coupling : Knowing the capacitive coupling of the gates to two donors, it is possible to couple donor atoms with the exchange interaction by setting them at the same electrochemical potential. Thus, automatically determining the capacitive coupling of gates is a valuable step towards donor coupling, a key element of quantum computing.

Chapter 6

Concluding Remarks

With respect to the thesis goals, automated identification and tracking of transitions, the developed solution has been significantly successful. `find_transitions()` has been shown to be an effective and efficient method to automatically identify transitions within a given charge stability diagram. `find_transitions_3D()` and `track_transitions_single()` were also shown to be highly effective at automatically tracking the movement of a single transition while varying a third slow gate. While `track_transitions_multi()` has exhibited critical limitations, Section 3.3 demonstrated it is a substantial step in the direction of simultaneously tracking multiple transitions. `track_transitions_multi()` was even used to automatically generate the values of a capacitance matrix for two donors at once.

The work done in this thesis will not only provide researchers with tools they can use now, but it has made valuable progress for donor triangulation, automated tuning, and donor coupling.

Ultimately, this thesis has had great success and will be a valuable contribution to the research done at CQC2T.

Bibliography

[1] M. Roser, "Technological progress." Available at https://ourworldindata.org/technological-progress (2018/05/27).

[2] A. D. Elster, "Zeeman splitting." Available at http://www.mriquestions.com/energy-splitting.html (2018/05/27).

[3] F. Kling, "Tunnel effekt." Available at https://commons.wikimedia.org/wiki/File:TunnelEffektKling1.png (2018/05/27).

[4] A. M. et al, "Architecture for high-sensitivity single-shot readout and control of the electron spin of individual donors in silicon," *Physical Review B*, vol. 80, August 2009.

[5] A. M. et al, "Single-shot readout of an electron in silicon," *NATURE*, vol. 467, pp. 687–691, October 2010.

[6] G. E. Moore, "Cramming more components onto integrated circuits," *Electronics Magazine*, vol. 38, p. 3, April 1965.

[7] M. Owen, "Apple's 'a12' chip reportedly in production using 7nm process from tsmc." Available at http://appleinsider.com/articles/18/04/23/apples-a12-chip-production-using-7nm-process-in-second-half-of-2018-may-help-tsmc (2018/05/27).

[8] B. Schumacher *Phys. Rev. A*, vol. 51, p. 2738–2747, April 1995.

[9] S. Hill and W. K. Wootters *Physical review letters*, vol. 78, no. 26, p. 5022–5025, 1997.

[10] R. Feynman *International journal of theoretical physics*, vol. 21, no. 6, pp. 467–488, 1982.

[11] C. H. Bennett and D. P. DiVincenzo *NATURE*, vol. 404, pp. 247–255, March 2000.

[12] R. Mohsen, "Quantum theory of tunnelling.," *World Scientific*, vol. 4, 2003.

[13] J. J. Pla, *Single Atom Spin Qubits in Silicon*. PhD thesis, 2013.

[14] S. Angus, A. Ferguson, A. Dzurak, and R. Clark, "Gate-defined quantum dots in intrinsic silicon," *Nano Letters*, vol. 7, no. 7, p. 2051–2055, 2007.

[15] D. P. DiVincenzo *Fortschr. der Phys.*, vol. 48, pp. 771–783, 2000.

[16] P. E. Hart and O. R. Duda, "Use of the hough transformation to detect lines and curves in pictures," *Comm. ACM*, vol. 15, pp. 11–15, 1972.

[17] I. Sobel, "An isotropic 3x3 image gradient operator," 2014.

Appendix A - Test Results

Decimation

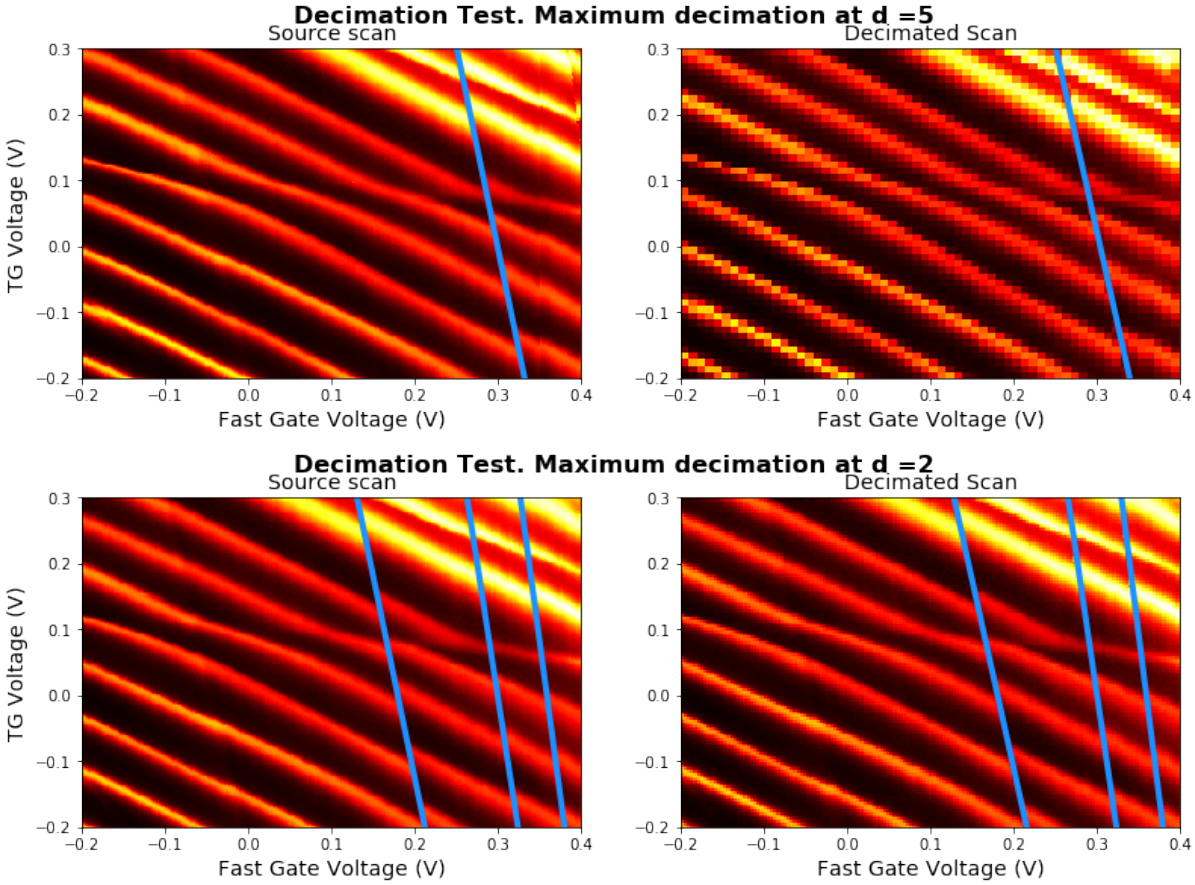


Figure 1: Decimation tests 1. Images generated in Python.

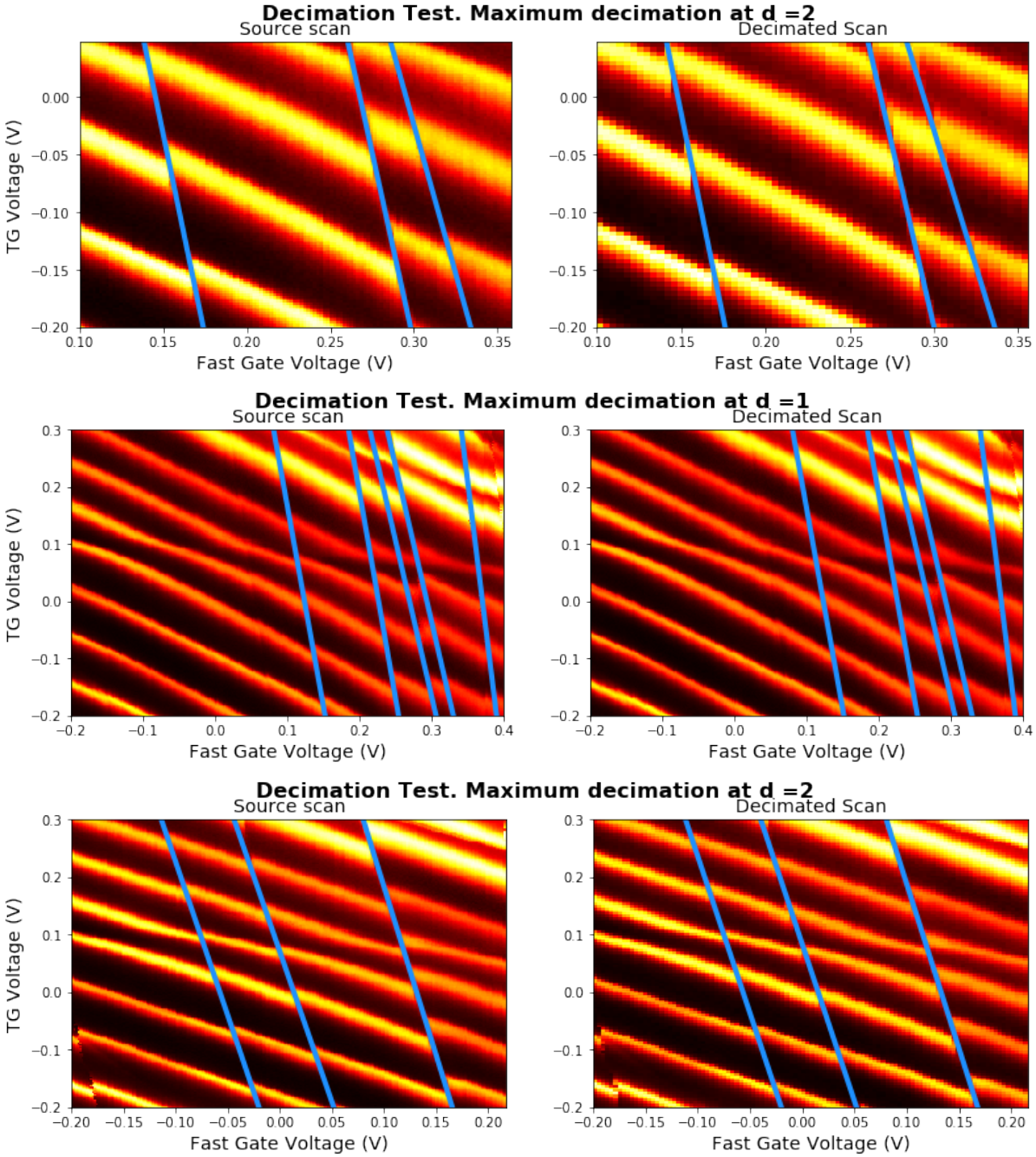


Figure 2: Decimation tests 2. Images generated in Python.

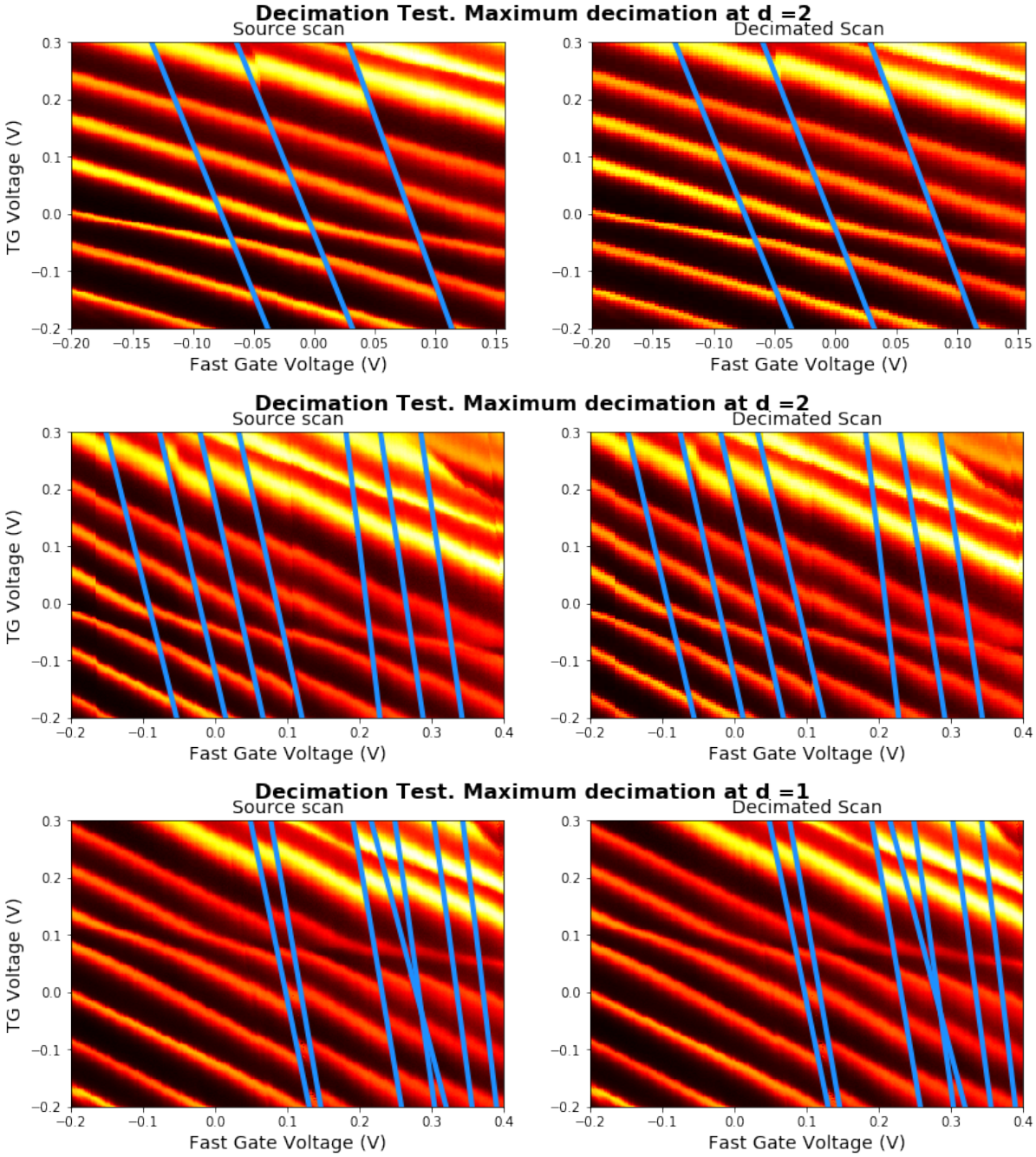


Figure 3: Decimation tests 3. Images generated in Python.

Noise

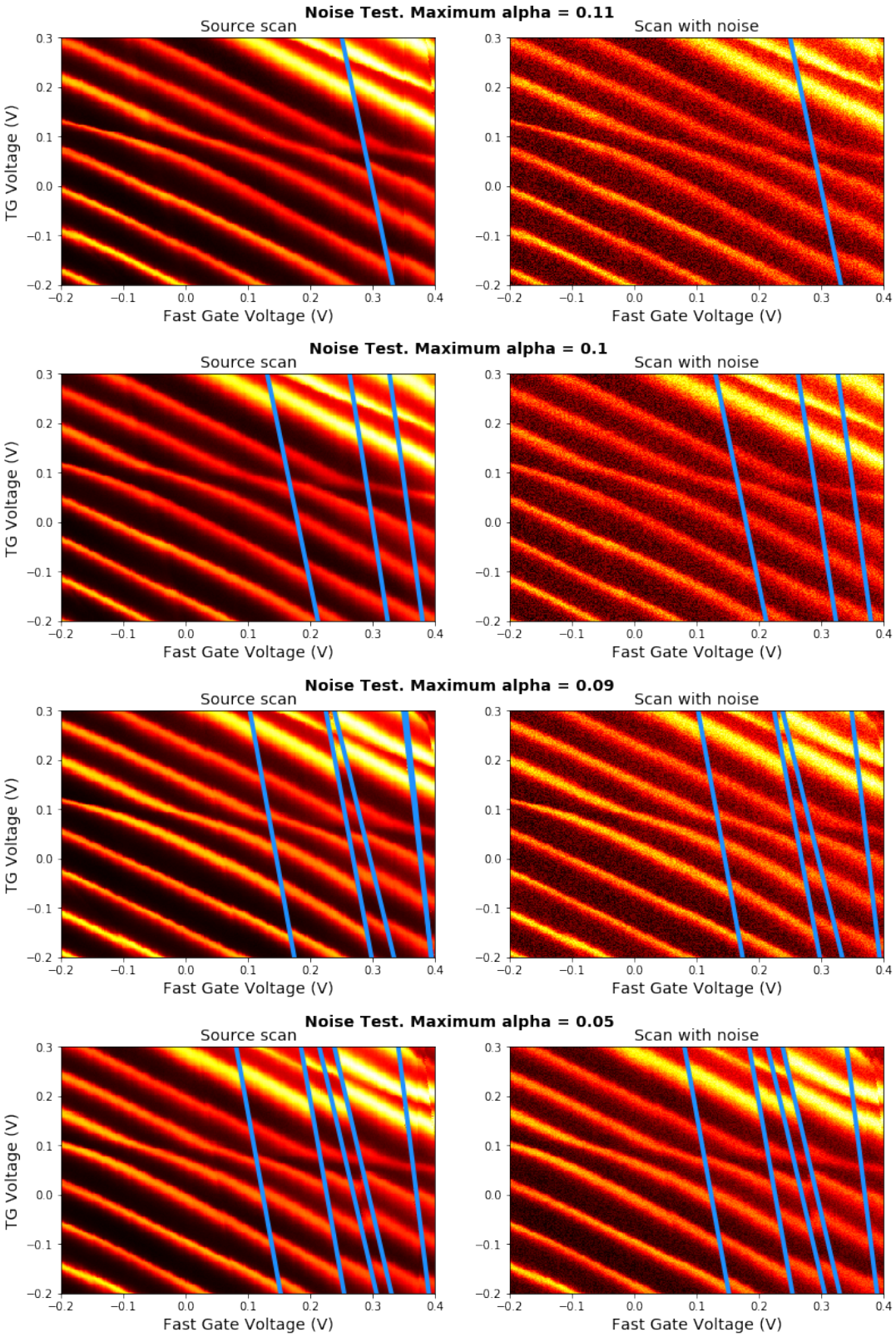


Figure 4: Noise tests 1. Images generated in Python.

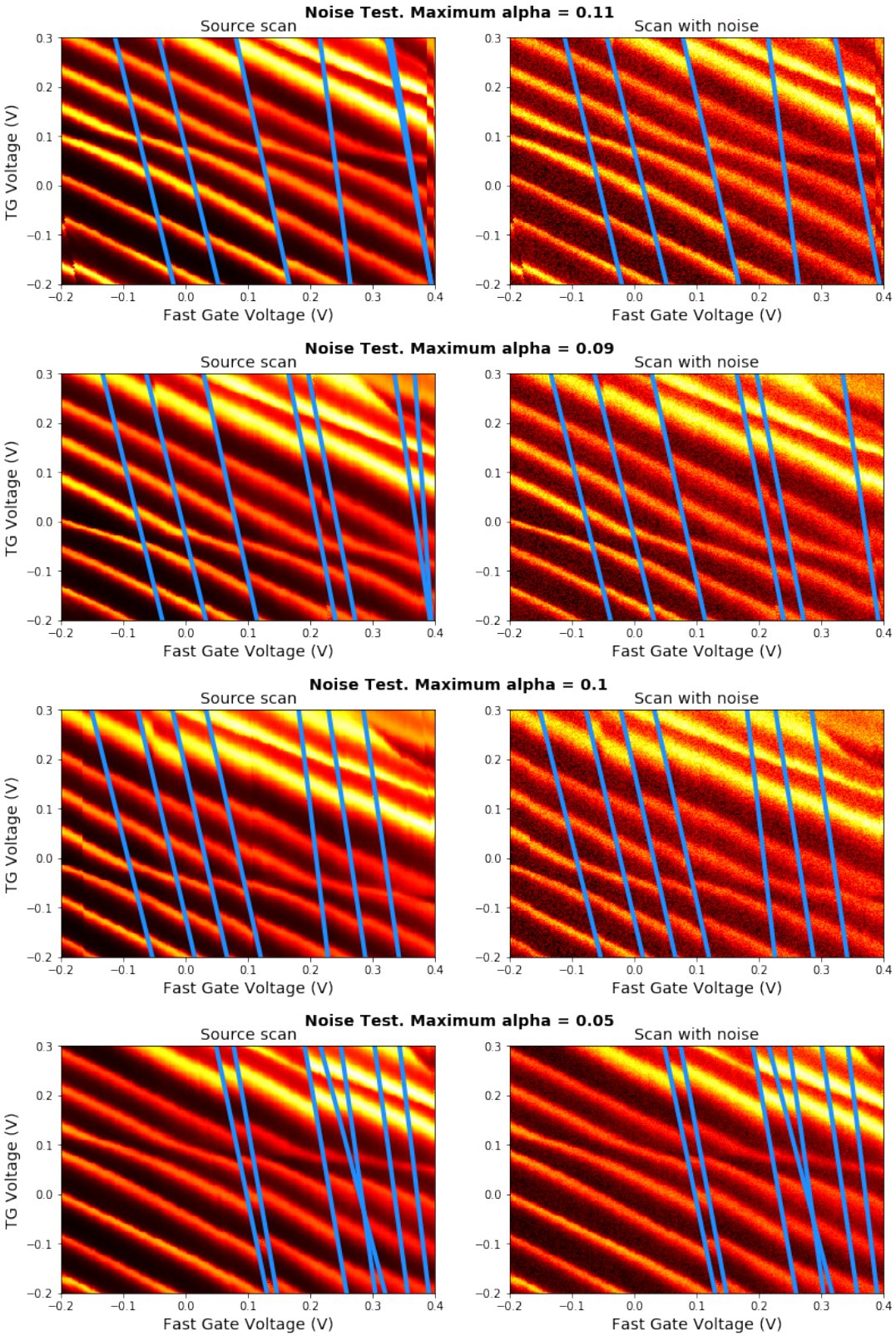


Figure 5: Noise tests 2. Images generated in Python.

Speed

A number of charge stability diagram were tested on `find_transitions()` with varying size and number of transitions. The timing results are shown in the table below:

Num pixels	Num transitions	Time (ms)
75551	7	1570
75551	3	821
75551	5	1090
75551	5	1220
75551	7	1480
75551	6	1270
75551	7	1420
75551	7	1610
37650	3	593
37650	3	530
37650	5	681
37650	4	751
37650	6	884
37650	3	539
37650	5	641
37650	3	612
18750	3	219
18750	3	227
18750	5	262
18750	5	262
18750	6	312
18750	4	208
18750	4	223
18750	3	246

Accuracy

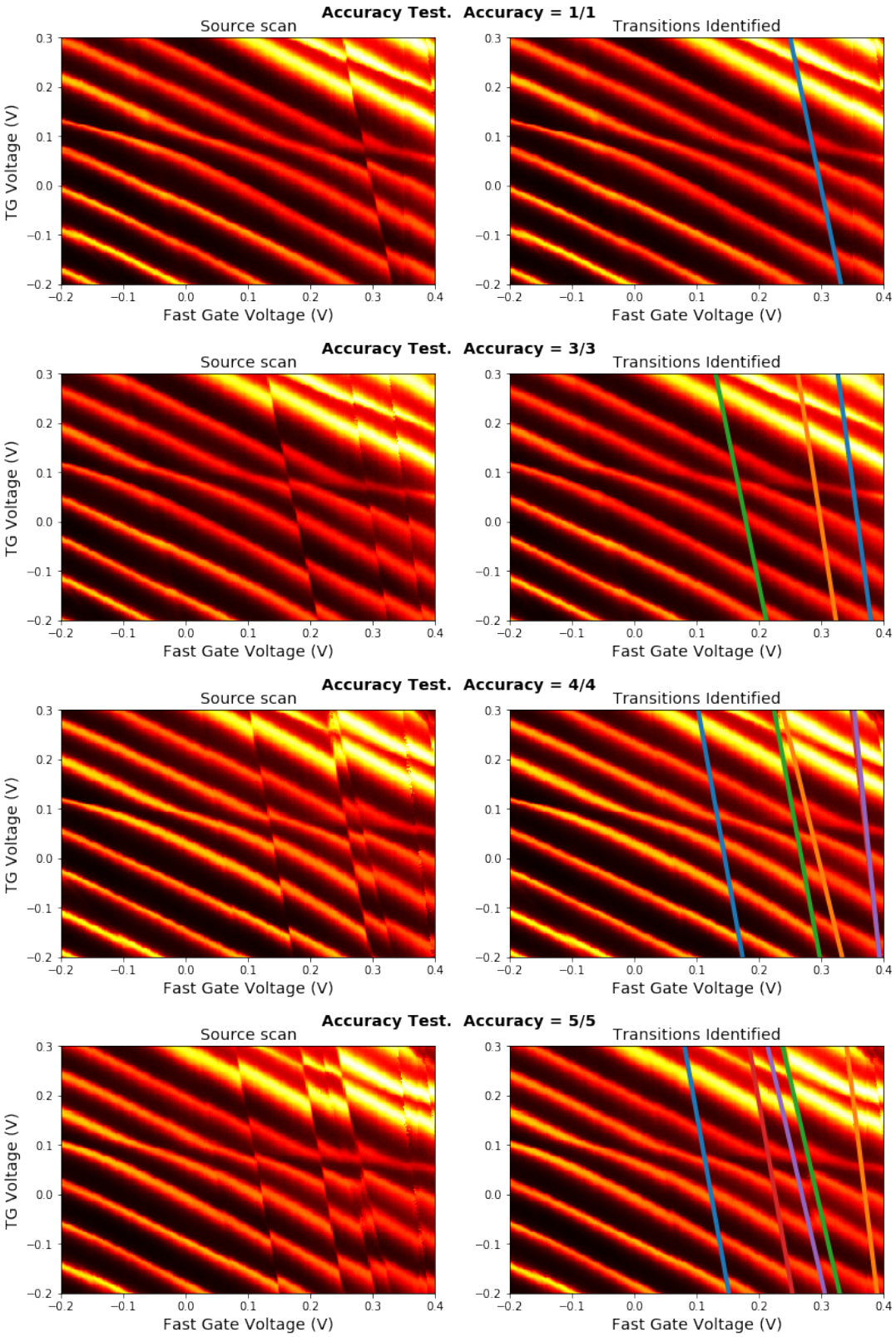


Figure 6: Accuracy tests 1. Images generated in Python.

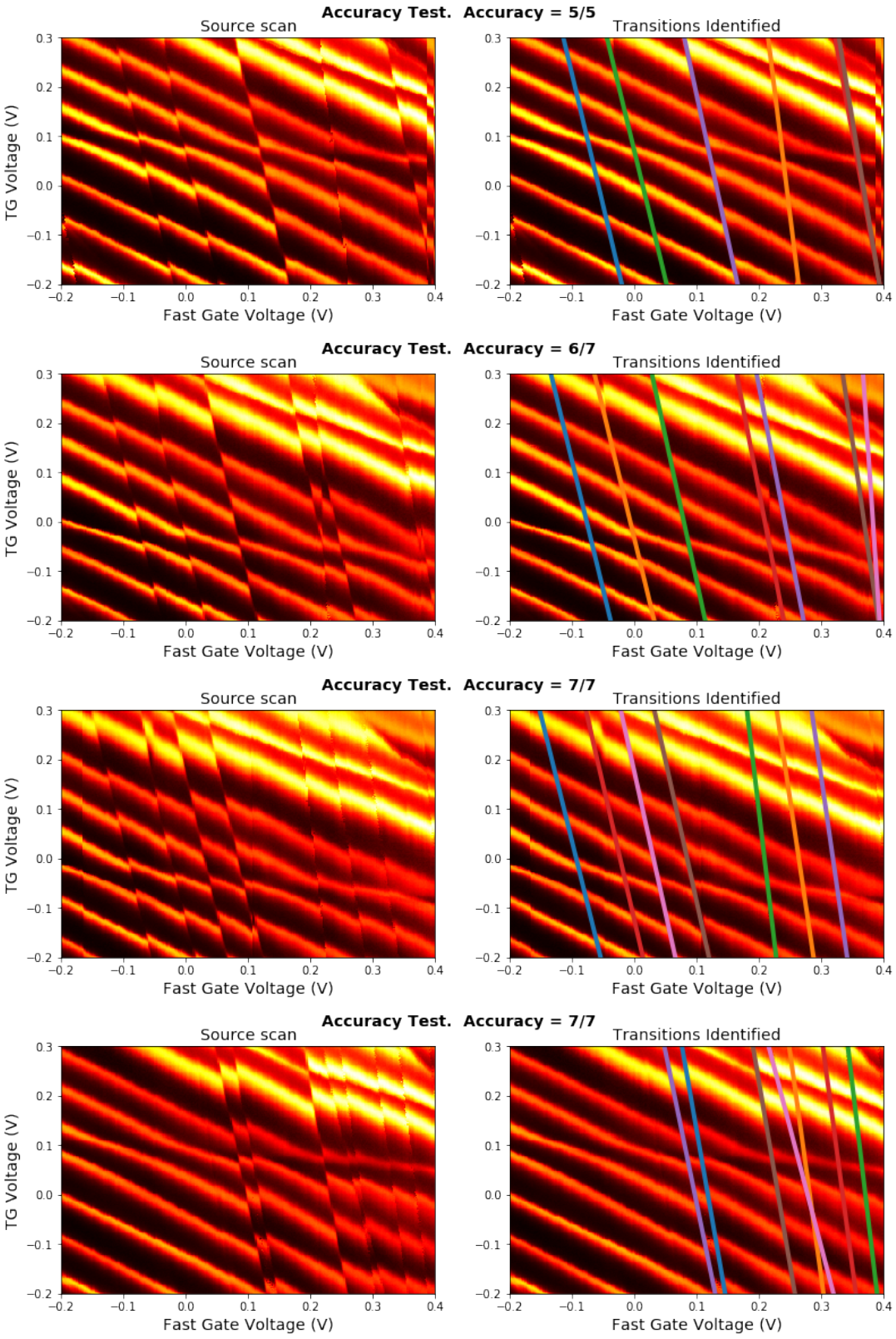


Figure 7: Accuracy tests 2. Images generated in Python.

Appendix B - Solution Code

Below is a Python library containing all the code written for the solution of this thesis. Source code and demonstration code are also available at:

https://github.com/dennisotter/D-Otter-Honours-Thesis.

```
### Automated Tracking of Single Atom Qubits for Silicon Quantum Computing ###
# By Dennis Otter
# Honours Thesis 2018.
#
# If you would like an in-depth explaination of this code, please refer to my thesis paper.
# The paper also includes notes on some vital improvements to this code.

from typing import List, Tuple
import numpy as np
from scipy import signal
from scipy.signal import convolve2d
from math import pi
from matplotlib import pyplot as plt
from qcodes import load_data, MatPlot
import itertools as it

def max_index(M: np.ndarray) -> Tuple[int, int]:
    """Returns the index of the maximum element in M.

    Args:
        M: n-dimensional matrix. Ideally a 2-dimensional theta matrix,
            2-dimensional charge stability diagram, or 2-dimensional
            Hough transform matrix.

    Returns:
        Array with x and y index of maximum element.
        For a transition gradient matrix, I[0] is dx, and I[1] is x1.
    """
    return np.unravel_index(np.argmax(M), M.shape)


def calculate_theta_matrix(Z: np.ndarray, filter: bool = False) -> np.ndarray:
```

```
"""Computes the theta matrix for a 2-dimensional charge stability diagram.

The theta matrix indicates the direction of the 2-dimensional gradient.

# TODO Fine-tuning these values could definitely improve performance.
# TODO We are applying a kernel, so perhaps it makes sense to have kernel_size as a keywor

Args:
    Z: 2-dimensional charge stability diagram matrix.
    filter: Enables filtering during the calculations.

Returns:
    theta: 2-dimensional theta matrix.
"""

### Filter coefficients

# Refer to https://en.wikipedia.org/wiki/Sobel_operator
# That explains the prinicples used here.

# Sobel Operator
# SY and SX are differentiating kernels, while the ySfil and xSfil are averaging.
SY = np.array([[1], [0], [-1]])
ySfil = np.array([[1, 2, 1]])
SY = convolve2d(SY, ySfil)

SX = np.array([[1, 0, -1]])
xSfil = np.array([[1], [2], [1]])
SX = convolve2d(SX, xSfil)

# Binomial filter kernel for the X and Y gradient matrices.
xGfil = np.array([[1], [2], [1]])
yGfil = np.array([[1, 2, 1]])
Gfil = convolve2d(xGfil, yGfil)

# Binomial filter kernel for the source matrix Z prior to computing the gradients.
xZfil = np.array([[1], [2], [1]])
yZfil = np.array([[1, 2, 1]])
Zfil = convolve2d(xZfil, yZfil)

if filter:
    # This will filter the source matrix Z prior to computing the gradients.
    Z = convolve2d(Z, Zfil, mode='valid')

#Calculate X and Y gradient matrices
GY = convolve2d(Z, SY, mode='valid')
GX = convolve2d(Z, SX, mode='valid')

if filter:
    #This will filter the gradient matrices once they have been calculated.
    GY = convolve2d(GY, Gfil, mode='valid')
```

```
        GX = convolve2d(GX, Gfil, mode='valid')

    #Calculate gradient direction.
    theta = np.arctan(GY / GX)
    return theta


def find_matrix_mode(M: np.ndarray, bins: int = 100) -> float:
    """Determines the mode of a matrix (most-often occurring element).

    Mode is found by first generating a histogram of matrix values, and then
    returning the center value of the bin with the highest count.
    Values are grouped because floating numbers are only approximately equal.

    Args:
        M: n-dimensional matrix. Ideally a 2-dimensional theta matrix.

    Returns:
        mode: most common element of M after grouping via a histogram.
    """

    hist, hist_edges = np.histogram(M, np.linspace(-pi, pi, bins))
    ind = max_index(hist)
    mode = (hist_edges[ind] + hist_edges[ind + np.array([1])]) / 2
    return mode[0]


def calculate_hough_transform(theta_dif: np.ndarray) -> np.ndarray:
    """Compute the Hough transform matrix from a given theta matrix.

    # TODO Have a variable input for the gradient range to scan for. This is dx_max

    Args:
        theta_dif: Filtered 2-dimensional theta matrix of a charge stability diagram.

    Returns:
        2-dimensional Hough transform matrix.
            x-axis is start position, y-axis is gradient.
    """

    # Generate Lines
    len_y, len_x = theta_dif.shape
    y_line = np.arange(len_y, dtype=int)

    # This value was found to be roughly twice the maximum dx value a transition will have
    # Essentially this is the minimum gradient
    dx_max = int(np.ceil(len_y / 3))

    hough = np.zeros((dx_max, len_x))

    for x1 in range(len_x):
```

```
        for dx in range(min([x1 + 1, dx_max])):
            x_line = x1 + np.round(-dx * y_line / len_y).astype(int)
            hough[dx, x1] = np.mean(theta_dif[y_line, x_line])

    return hough


def delete_transition(theta_dif: np.ndarray, hough_filt: np.ndarray, hough_raw: np.ndarray) ->
    """Removes a transition from a theta_dif matrix. In order to find transitions, they are id
    The most prominent transition is identified first, then removed from the theta_dif matrix
    prominent transition can be found.

    Args:
        theta_dif: 2-dimensional filtered theta matrix of a charge stability diagram.
        location: Base index of the charge transfer event in Z
        gradient: Gradient of the charge transfer event in Z

    Returns:
        theta_dif: modified filtered 2-dimensional theta matrix, with the specified transition
    """

    dx, location = max_index(hough_filt)
    len_y,len_x = theta_dif.shape
    y_line = np.arange(len_y, dtype=int)
    dx_max = hough_filt.shape[0]

    # Start and stop are the base locations from which to delete a transition from.
    start = location
    stop = location
    while((hough_filt[dx,start] >= 0.25)&(start>0)): #this value 0.25 needs to be same as in f
        start -=1
    while((hough_filt[dx,stop] >= 0.25)&(stop<(len_x-1))):
        stop +=1

    for x1 in range(start, stop):
        x_line = x1 + np.round(-dx * y_line / len_y).astype(int)
        theta_dif[y_line, x_line] = 0

    x1 = np.array(range(start-dx,min([stop+dx_max-dx, len_x])))
    for i in range(x1.size):
        for dx_scan in range(min([x1[i] + 1, dx_max])):
            x_line = x1[i] + np.round(-dx_scan * y_line / len_y).astype(int)
            hough_raw[dx_scan, x1[i]] = np.mean(theta_dif[y_line, x_line])



    return theta_dif, hough_raw


def calculate_theta_dif(theta: np.ndarray, theta_mode: float) -> np.ndarray:
    """Calculates the difference between the values of a given
```

```
    theta matrix, and theta_mode. This function is used
    to highlight pixels that are likely to lie on a transition.

    Args:
        theta: 2-dimensional theta matrix of a charge stability diagram.
        theta_mode: Modal theta value, found with find_matrix_mode().

    Returns:
        theta_dif: modified 2-dimensional theta matrix,
            with the specified transition removed.
    """
    #you can change this method for potential improvements
    theta_dif = 1 - np.cos(theta_mode - theta) ** 2
    return theta_dif


def plot_transitions(x: np.ndarray, y: np.ndarray, Z: np.ndarray, transitions: List[dict]):
    """Plots the source charge stability diagram, next to a charge
    stability diagram with the given transitions plotted.

    Args:
        x: 1-dimensional voltage vector for the x-axis of Z
        y: 1-dimensional voltage vector for the y-axis of Z
        Z: 2-dimensional charge stability diagram matrix.
        transitions: List of transitions found using find_transitions()
    """
    fig0,(ax0,ax1) = plt.subplots(1, 2, figsize=[13,4])
    fig0.suptitle('Transition Identification', fontsize=16, fontweight='semibold')

    ax0.pcolormesh(x, y, Z, cmap='hot')
    ax0.set_xlabel('Fast Gate Voltage (V)', fontsize=14)
    ax0.set_ylabel('TG Voltage (V)', fontsize=14)
    ax0.set_title('Source scan', fontsize=16)

    ax1.pcolormesh(x, y, Z, cmap='hot')
    ax1.set_xlabel('Fast Gate Voltage (V)', fontsize=14)
    ax1.set_title('Transitions Identified', fontsize=16)

    yvals = ax1.get_ylim()
    for transition in transitions:
        x_base = transition['location']
        if (type(x_base) is int) : x_base = x[x_base]

        xvals = [x_base, x_base]
        xvals[1] += (yvals[1] - yvals[0]) / transition['gradient']
        ax1.plot(xvals, yvals, '-', linewidth=4)
    plt.show()


def plot_hough_transform(hough: np.ndarray, theta_dif: np.ndarray, location: int = -1, dx: int
    """Plots a filtered theta matrix next to its Hough Transform.
```

```
    Args:
        hough: 2D hough transform matrix
        theta_dif: 2D filtered theta matrix.
    """
    fig, axes = plt.subplots(1, 2, figsize=[13,4])

    c = axes[1].pcolormesh(hough, cmap='inferno',vmin=0,vmax=vmax)
    axes[1].set_ylabel('dx value', fontsize=14)
    axes[1].set_xlabel('Fast Gate Voltage index', fontsize=14)
    axes[1].set_title('Hough Transform Matrix', fontsize=16)

    axes[0].pcolormesh(theta_dif, cmap='gray')
    axes[0].set_xlabel('Fast Gate voltage index', fontsize=14)
    axes[0].set_ylabel('TG voltage index', fontsize=14)
    axes[0].set_title('Filtered Theta Matrix', fontsize=16)

    if location is not -1:
        axes[1].scatter(location, dx, marker= 'o', linewidth=4, color='magenta')
        yvals = axes[0].get_ylim()
        xvals = [location, location-dx]
        axes[0].plot(xvals, yvals, '--', linewidth=2,color='magenta')

    plt.show()


def find_transitions(x: np.ndarray,
                     y: np.ndarray,
                     Z: np.ndarray,
                     true_units: bool = True,
                     plot = False) -> List[dict]:
    """Locate transitions within a 2-dimensional charge stability diagram

    Args:
        x: 1-dimensional voltage vector for the x-axis of Z
        y: 1-dimensional voltage vector for the y-axis of Z
        Z: 2-dimensional charge stability diagram matrix.
        true_units:
            if True:
                Where applicable, return all values in proper units. i.e. voltage and current.
            if False:
                Return values in calculation specific form. i.e. index and ratios.
        charge_transfer:
            Enables calculation of voltage and current shift information about transitions.
            This is required to calculate dV, dI, dI_x, dI_y
        plot:
             - 'Off'     = No plots
             - 'Simple'  = Plot of DC data and transition data next to it
             - 'Complex' = All of simple, plus the transition_gradient and theta plots for eac

    Returns: a list of dictionaries, one entry for each transition found:
```

```
# TODO (Serwan) simplify this part
if true_units == True:
    location  (float): Voltage at the base of the transition.
    gradient  (float): Gradient of the transition. in y_Voltage/x_Voltage
    intensity (float): Value between 0 and 1, indicating the strength of the transition
    dV        (float): The shift of coulomb peaks from a charge transfer event. dV = dVtop
if true_units == False):
    location    (int): Index at the base of the transition.
    gradient  (float): Gradient of the transition. in y-index/x-index
    intensity (float): Value between 0 and 1, indicating the strength of the transition
    dV          (int): The shift of coulomb peaks from a charge transfer event in terms of
                       dV*(y[1]-y[0]) = dVtop = d_q/Ctop
"""

theta = calculate_theta_matrix(Z, filter=True)
theta_mode = find_matrix_mode(theta)
theta_dif = calculate_theta_dif(theta,theta_mode)

hough_raw = calculate_hough_transform(theta_dif)
H_filter = np.ones((3,3))
hough_filt = convolve2d(hough_raw, H_filter, mode='same')/9

transitions = []

# This condition seems good now, could be improved later but i'd say low priority.
while ((np.max(hough_filt) > 3*np.mean(hough_filt)) & (np.max(hough_filt) >0.25)):

    #maximum element of transition_gradient will reveal where the transition is
    raw_gradient, raw_location = max_index(hough_filt)
    intensity = np.max(hough_filt)

    if (plot == 'Complex'): plot_hough_transform(hough_filt, theta_dif,
                                                 raw_location, raw_gradient,
                                                 vmax= intensity)

    # When filtering with convolution, the size of theta and transition_gradient will diff
    # The following lines adjust the raw_location from transition_gradient to be a true lo
    difx = (x.shape[0] - theta.shape[1]) / 2 #difference in x-axis size
    dify = (y.shape[0] - theta.shape[0]) / 2 #difference in y-axis size
    #Adjusting the location:
    # raw_location
    # + difference in x
    # + difference in x from dify due to gradient shift.
    location = int(difx + raw_location + np.ceil(dify * raw_gradient / theta.shape[0]))

    #Recalculate theta and hough with the identified transition removed
    theta_dif, hough_raw = delete_transition(theta_dif, hough_filt, hough_raw)
    hough_filt = convolve2d(hough_raw, H_filter, mode='same')/9

    #If the gradient registers as being close to perfectly vertical, skip over this transi
    #since transitions are never perfectly vertical.
```

```
        #You can change this if you don't believe me but the algorithm will be more buggy
        if(raw_gradient <2): continue

        #gradient = dy/dx = y_length/dx = theta.shape[0]/raw_gradient
        gradient = -(theta.shape[0]/raw_gradient)

        # The following lines combine together to come up with the final error calculation
        #minimum_gradient = dy/max_dx = -(theta.shape[0]/(raw_gradient+1))
        #maximum_gradient = dy/min_dx = -(theta.shape[0]/(raw_gradient-1))
        #abs_error        = (maximum_gradient - minimum_gradient)/2
        #percent_error    = abs_error/observed_gradient*100                = abs_error/(-(thet
        gradient_error    = (np.abs(raw_gradient/(raw_gradient-1)-1) + np.abs(raw_gradient/(ra

        dV = get_charge_transfer(Z, location, gradient, theta_mode)
        if true_units:  # Convert indices to units
            gradient = gradient * (y[1] - y[0]) / (x[1] - x[0])  # in V/V
            location = x[location]  # units in V
            dV = dV * (y[1] - y[0]) # units in V

        #Add transition entry onto the output list
        transitions.append({'location': location,
                    'gradient': gradient,
                    'gradient_error': gradient_error,
                    'intensity': intensity,
                    'dVtop': dV})

    if (plot == 'Complex'): plot_hough_transform(hough_filt,theta_dif,vmax=1)

    if (plot == True)|(plot == 'Complex'): plot_transitions(x,y,Z,transitions)

    return transitions


def get_charge_transfer(Z: np.ndarray,
                                    location: int,
                                    gradient: float,
                                    theta_mode: float) -> int:
    """Calculates how much the coulomb peaks shift at a charge transfer event.
    It does this by taking two slices either side of the transition, then comparing the shift.

    Args:
        Z         : 2-dimensional charge stability diagram matrix.
        location  : Base index of the charge transfer event in Z
        gradient  : Gradient of the charge transfer event in Z
        theta_mode: Mode of theta (most common theta value)

    Returns:
        dV: The shift of coulomb peaks from a charge transfer event.
                    Given as a shift of index in Z. When properly scaled:
                        dV = dVtop = d_q/Ctop
    """
```

```
    ly = Z.shape[0]
    yl = np.arange(ly, dtype=int)
    xl = (location + np.round(yl / gradient)).astype(int)


    # Take two current lines to the left and right of the transition, these will be compared t
    shift = 4
    pre  = xl - shift
    post = xl + shift
    if ((min(pre) < 0)|(max(post)>=Z.shape[1])):
        #if the pre-post lines are out of bounds, then don't bother computing. This could be i
        return -1
    line_pre = Z[yl, pre]
    line_pos = Z[yl, post]

#     Average Magnitude Difference Function.
#     This will shift and compare the lines before and after to see how much the transition sh
    AMDF = np.zeros(ly)
    for i in range(ly):

        AMDF[i] = -np.mean(np.abs(
            line_pre[np.array(range(0, ly - i))]
            -line_pos[np.array(range(i, ly))]))  \
            * (1+i)/(1)                     #Adjustment for the decreasing comparison window

    # peakshift exists to find out how much of the shift in coulomb peaks in the difference is
    # natural gradient of the coulomb peaks. This can be worked out using tan, the coulomb pea
    # and the shift amount
    peakshift = np.round(
        np.abs(np.tan(theta_mode - np.pi/2)) *(1 +2*shift)).astype(
        int)

    dV = max_index(AMDF)[0] + peakshift

    return dV.astype(int)#, dI, dI_x, dI_y

## Start code for 3D tracking

def find_transitions_3D(slow: np.ndarray,
                        fast: np.ndarray,
                        TG: np.ndarray,
                        Z: np.ndarray,
                        plot: bool = False) -> List[List[dict]]:
    """Locate transitions within a 3-dimensional charge stability diagram

    Args:
        slow: 1-dimensional voltage vector for the slow gate voltage of Z
        fast: 1-dimensional voltage vector for the fast gate voltage of Z
        TG  : 1-dimensional voltage vector for the TG voltage of Z
        Z   : 3-dimensional charge stability diagram matrix. [slow,fast,TG]
        plot:
```

```
            - False = No plots
            - True  = Plots the 3D transitions.

    Returns: A list of transition lists over the third dimension found with find_transitions()
    """
    transition_list = []
    for i in range(Z.shape[0]):
        transition_list.append(find_transitions(fast,TG, Z[i,:,:], true_units=True))

    if plot:
        plot_transitions_3D(slow,fast,TG,Z,transition_list)
    return transition_list


def plot_slices_3D(slow: np.ndarray,
                   fast: np.ndarray,
                   TG: np.ndarray,
                   Z: np.ndarray,
                   transition_list: List[List[dict]]):
    """Plots 2D slices from a 3D charge stability diagram.

    Args:
        slow: 1-dimensional voltage vector for the slow gate voltage of Z
        fast: 1-dimensional voltage vector for the fast gate voltage of Z
        TG  : 1-dimensional voltage vector for the TG voltage of Z
        Z   : 3-dimensional charge stability diagram matrix. [slow,fast,TG]
        transition_list: A list of transition lists. Calculated with find_transitions_3D()
    """

    fig,(ax0,ax1,ax2) = plt.subplots(1, 3, figsize=[10,2.5])

    ind = 0
    ax0.pcolormesh(fast, TG, Z[ind,:,:], cmap='hot')
    ax0.set_xlabel('Fast Gate Voltage (V)', fontsize=14)
    ax0.set_ylabel('TG Voltage (V)', fontsize=14)
    ax0.set_title('Slow Gate = '+str(slow[ind])+'V', fontsize=14, fontweight='semibold')

    if len(transition_list[ind]) is not 0:
        x_base = transition_list[ind][0]['location']
        yvals = ax0.get_ylim()
        xvals = [x_base, x_base+ (yvals[1] - yvals[0])/transition_list[ind][0]['gradient']]
        ax0.plot(xvals, yvals, linewidth=6,color='lawngreen')


    ind = np.floor(slow.shape[0]/2).astype(int)
    ax1.pcolormesh(fast, TG, Z[ind,:,:], cmap='hot')
    ax1.set_xlabel('Fast Gate Voltage (V)', fontsize=14)
    ax1.yaxis.set_ticklabels([])
    ax1.set_title('Slow Gate = '+str(slow[ind])+'V', fontsize=14, fontweight='semibold')

    if len(transition_list[ind]) is not 0:
```

```
        x_base = transition_list[ind][0]['location']
        yvals = ax1.get_ylim()
        xvals = [x_base, x_base+ (yvals[1] - yvals[0])/transition_list[ind][0]['gradient']]
        ax1.plot(xvals, yvals, linewidth=6,color='magenta')


    ind = -1
    ax2.pcolormesh(fast, TG, Z[ind,:,:], cmap='hot')
    ax2.set_xlabel('Fast Gate Voltage (V)', fontsize=14)
    ax2.yaxis.set_ticklabels([])
    ax2.set_title('Slow Gate = '+str(slow[ind])+'V', fontsize=14, fontweight='semibold')

    if len(transition_list[ind]) is not 0:
        x_base = transition_list[ind][0]['location']
        yvals = ax2.get_ylim()
        xvals = [x_base, x_base+ (yvals[1] - yvals[0])/transition_list[ind][0]['gradient']]
        ax2.plot(xvals, yvals, linewidth=6,color='aqua')


def plot_transitions_3D(slow: np.ndarray,
                        fast: np.ndarray,
                        TG: np.ndarray,
                        Z: np.ndarray,
                        transition_list: List[List[dict]],
                        fit_list: List[dict] = None,
                        slices = False):
    """Plots the transitions found over a 3D scan with find_transitions_3D().

    Args:
        slow: 1-dimensional voltage vector for the slow gate voltage of Z
        fast: 1-dimensional voltage vector for the fast gate voltage of Z
        TG  : 1-dimensional voltage vector for the TG voltage of Z
        Z   : 3-dimensional charge stability diagram matrix. [slow,fast,TG]
        transition_list: A list of transition lists. Calculated with find_transitions_3D()
        fit_list: A list of fit lines, linking the transitions in transition_list.
            fit_list is calculated with track_transitions_multi or track_transitions_single.
        slices: if True, then 2D slices will be plotted above the tracked diagram.
    """
    if slices:
        plot_slices_3D(slow, fast, TG, Z, transition_list)

    n_slow = Z.shape[0]
    x_points = []
    y_points = []
    grad_points = []
    for i in range(n_slow):
        for T in transition_list[i]:
            fast_val = T['location']
            x_points.append(slow[i])
            y_points.append(fast_val)
            grad_points.append(T['gradient'])
```

```
    fig,ax = plt.subplots(1, 1, figsize=[10,5])
    plot = ax.scatter(x_points, y_points, c=grad_points)
    if slices is False:
        c = fig.colorbar(plot, ax=ax)
        c.set_label('TG/Fast Gradient', fontsize=14)

    xvals = np.array([slow[0], slow[-1]])
    if fit_list is not None:
        for F in fit_list:
            yvals = xvals*F['fast/slow gradient'] +F['fast intercept']
            ax.plot(xvals,yvals)#,color='red')

    xmargin = (np.max(x_points)-np.min(x_points))/13
    xlim = [(np.min(x_points)-xmargin), (np.max(x_points)+xmargin)]
    ymargin = (np.max(y_points)-np.min(y_points))/10
    ylim = [(np.min(y_points)-ymargin), (np.max(y_points)+ymargin)]
    ax.set_xlim(xlim)
    ax.set_ylim(ylim)
    ax.set_ylabel('Fast Gate Voltage (V)', fontsize=14)
    ax.set_xlabel('Slow Gate Voltage (V)', fontsize=14);
    ax.set_title('3D Transition Plot', fontsize=14, fontweight='semibold')

    if slices:
        ind = 0
        ax.scatter(slow[ind],transition_list[ind][0]['location'],linewidth=20,color='lawngreen
        ind = np.floor(slow.shape[0]/2).astype(int)
        ax.scatter(slow[ind],transition_list[ind][0]['location'],linewidth=20,color='magenta')
        ind = -1
        ax.scatter(slow[ind],transition_list[ind][0]['location'],linewidth=20,color='aqua')


def track_transitions_single(slow: np.ndarray,
                     fast: np.ndarray,
                     TG: np.ndarray,
                     Z: np.ndarray,
                     transition_list: List[List[dict]],
                     plot: bool = False) -> List[dict]:
    """Tracks the transitions found with find_transitions_2D().

    There must be a single transition over the entire 3D scan for this to work.
    Any more or less will cause the algorithm to work incorrectly.
    This algorithm is very reliable when used appropriately.

    Args:
        slow: 1-dimensional voltage vector for the slow gate voltage of Z
        fast: 1-dimensional voltage vector for the fast gate voltage of Z
        TG  : 1-dimensional voltage vector for the TG voltage of Z
        Z   : 3-dimensional charge stability diagram matrix. [slow,fast,TG]
        transition_list: A list of transition lists. Calculated with find_transitions_2D()
```

```
        plot: If plot==True, then plot_transitions_2D() is automatically called with the respe
    """

    n_slow = Z.shape[0]
    x_points = []
    y_points = []
    grad_points = []
    for i in range(n_slow): #length of slow gate
        for T in transition_list[i]:
            x_points.append(i)
            y_points.append(T['location'])
            grad_points.append(T['gradient'])


    X = np.swapaxes(np.array((slow[x_points], np.ones((len(x_points)),dtype=int))),0,1)
    Y = np.asarray(y_points)

    r = np.linalg.lstsq(X,Y)[0]
    m_slow = r[0]
    b = r[1]
    m_fast = np.mean(grad_points)

    retval = []
    retval.append({'fast intercept': b,
              'fast/slow gradient': m_slow,
              'TG/fast gradient': m_fast,
              'TG/slow gradient': m_fast/m_slow})

    if plot:
        plot_transitions_3D(slow,fast,TG,Z,transition_list,retval)
    return retval


#The following code is unreliable. Do not use. Only included to demonstrate how it works for s
def track_transitions_multi(slow: np.ndarray,
                        fast: np.ndarray,
                        TG: np.ndarray,
                        Z: np.ndarray,
                        transition_list: List[List[dict]],
                        plot: bool = False) -> List[dict]:
    """Tracks the transitions found in find_transitions_3D().

    This will only work for any amount of transitions over the entire 3D scan
    However, this algorithm is not accurate; it can link transitions which are not correlated
    and it can miss transitions that are.

    *** WARNING: DO NOT USE THIS ALGORITHM FOR LARGE DATASETS ***
    This algorithm is incredibly inefficient.
    It calculates every possible combination of transitions and computes the least squares fit
    This relates to approx (n_transitions)^2*(n_scans)!
```

```
I have tested this for sets up to 6 scans (6 charge stability diagrams)
A test of 100 scans was sufficient to use over 16GB RAM and 100% of a 4-core processor.

Basically, don't use this function. I have included it only to demonstrate my attempt.
If you desire to correlate multiple transitions, use find_transitions_3D() with plot=True,
then visually decide which ones are correlated.

Args:
    slow: 1-dimensional voltage vector for the slow gate voltage of Z
    fast: 1-dimensional voltage vector for the fast gate voltage of Z
    TG  : 1-dimensional voltage vector for the TG voltage of Z
    Z   : 3-dimensional charge stability diagram matrix. [slow,fast,TG]
    transition_list: A list of transition lists. Calculated with find_transitions_3D()
    plot: If plot==True, then plot_transitions_3D() is automatically called with the respe
"""

n_slow = Z.shape[0]
points_list = []
for i in range(n_slow): #length of slow gate
    for T in transition_list[i]:
        points_list.append((i,
                            T['location'],
                            T['gradient']))

X_slow_0 = np.swapaxes(np.array((slow, np.ones((len(slow)),dtype=int))),0,1)
least_squares = []

combs = [] #combinations of points
for i in range((int)(n_slow*2/3), n_slow+1):
    combs.extend(list(it.combinations(range(n_slow),i))) #creates a list of possibilities

for X_slow in combs:
    slow_index = []
    for k in X_slow:
        slow_index.append(np.arange(len(transition_list[k])))
    slow_index = np.asarray(list(it.product(*slow_index)))

    for k in range(slow_index.shape[0]):

        Y_fast = [transition_list[X_slow[j]][slow_index[k,j]]['location'] for j in range(l
        G_fast = [transition_list[X_slow[j]][slow_index[k,j]]['gradient'] for j in range(l

        X_slow_1 = X_slow_0[X_slow,:]

        r = np.linalg.lstsq(X_slow_1,Y_fast)[0]
        m = r[0]
        b = r[1]
        Y_fit = X_slow_1@r
        if 0 in Y_fit:
            continue
        Y_dif = np.mean(np.abs(Y_fast/Y_fit-1))
```

```
        G_dif = np.max(np.abs((G_fast/np.mean(G_fast)-1)))

        Tot_dif = Y_dif/len(Y_fast)*(1+G_dif) #change this for different sorting

        lsq = {'m': m,
               'b': b,
               'Y_fast': Y_fast,
               'X_slow': X_slow,
               'Y_fit': Y_fit,
               'G_fast': G_fast,
               'Tot_dif': Tot_dif,}
        least_squares.append(lsq)

least_squares = sorted(least_squares, key=lambda ls: ls['Tot_dif'])
retval = []
numbrs = []


for L in least_squares:
    if (L['Tot_dif'] >0.005): break
    ok = True
    for Y in L['Y_fast']:
        if Y in numbrs:
            ok = False
            break
    if(ok):
        retval.append({'fast intercept': L['b'],
                  'fast/slow gradient': L['m'],
                  'TG/fast gradient': np.mean(L['G_fast']),
                  'TG/slow gradient': np.mean(L['G_fast'])/L['m']})
        numbrs.extend(L['Y_fast'])


if plot:
    plot_transitions_2D(slow,fast,TG,Z,transition_list, retval)

return retval
```