Chapter 1

Stitching IC Images

Michael Baumann[1], Chris Bose[2], Lorraine Dame[2], Isabelle Dechene[3], Maria Inez Goncalves[2], Robert Israel[1], Edward Keyes[4] (presenter), Frithjof Lutscher[5], David Lyder[6], Colin Macdonald[7], Jenny McNulty[8], Joy Morris[9], Nancy Ann Neudauer[10], Benjamin Ong[7], Robert Piché[11] (editor), Arūnas Šalkauskas[12], Karen Seyffarth[13], Brett Stevens[14], Tzvetalin Vassilev[15], Brian Wetton[1], Fabien Youbissi[16]

1.1 Introduction

Errors in stitching (or mosaicing) of integrated circuit (IC) images cause errors in the automated generation of the schematic. This increases costs and introduces delays as engineers must step in with interactive computer tools to correct the stitching. Our goal was to develop automated image stitching methods that keep stitching errors under $F/2$, where the minimum feature size F is around 10 pixels.

Although image stitching software is used in many areas such as photogrammetry, biomedical imaging, and even amateur digital photography, these algorithms require relatively large image overlap. For this reason they cannot be used to stitch the IC images, whose overlap is typically less than 60 pixels for a 4096 by 4096 pixel image.

[1]University of British Columbia
[2]University of Victoria
[3]McGill University
[4]Semiconductor Insights Inc.
[5]University of Alberta
[6]King's University College
[7]Simon Fraser University
[8]University of Montana
[9]University of Lethbridge
[10]Pacific University
[11]Tampere University of Technology
[12]Sorex Software Inc.
[13]University of Calgary
[14]Carleton University
[15]University of Saskatchewan
[16]Laval University

IC images are currently stitched as follows. First, the offset between each image and its neighbours is determined by minimising an error function that measures the difference between features in overlapping image pairs. These offsets are then used to stitch the images together in some predetermined pattern. Errors tend to accumulate as the images are added sequentially to the grid.

In Section 1.2 we use algorithmic graph theory to study optimal patterns for adding the images one at a time to the grid. In the remaining sections we study ways of stitching all the images simultaneously using different optimisation approaches: least squares methods in Section 1.3, simulated annealing in Section 1.4, and nonlinear programming in Section 1.5.

1.2 Stitching using Graph Theory

Here we present a graph theoretic approach to the problem. This approach is purely local in the sense that we only use the fact that any given pair of adjacent images can be stitched together perfectly, i.e., all features match, due to the offset data from pairwise correlation. At present, these local correlations are used for stitching following a certain pattern. We propose different patterns which should give better global results. To that end, we formulate the problem in a graph theoretic way and introduce some error measures. We find and compare several alternative stitching patterns which have smaller errors than the one currently used. Finally, we extend the approach to include information about the reliability of the pairwise correlation data.

1.2.1 Notation

We assume the images are arranged in a rectangular shape of m rows and n columns, where $m \geq n$. Define the graph G by letting each image correspond to a vertex. Two vertices are joined by an edge if the corresponding images are adjacent. Hence, G is an $m \times n$ grid graph. We denote by $E(G)$ the edges of G, and for adjacent $u, v \in G$ we write $uv \in E(G)$.

We use the stitching pattern to define a *spanning tree* T of G, i.e., a connected subgraph without cycles. An edge of G is an edge of T if the corresponding images are stitched together perfectly according to the offset values. The current method of row-wise stitching corresponds to the row pattern given by the solid lines in Figure 1.1. Here, the images are stitched together row-wise and then the strips are stitched together along a "spine"

We assume that the error between u and v with respect to the true offset values increases with the distance between u and v in the tree T. To quantify this error, we define the following. For each $uv \in E(G)$,

$$\begin{aligned}
d'_T(u,v) & \quad \text{denotes the distance between } u \text{ and } v \text{ in the tree } T; \\
d_T(u,v) &= d'_T(u,v) - 1; \\
S(T) &= \Sigma_{uv \in E(G)} d_T(u,v) \text{ (i.e., the sum of all the distances, in } T\text{, between} \\
& \quad\ \text{adjacent vertices of } G); \\
M(T) &= \max_{uv \in E(G)} d_T(u,v).
\end{aligned}$$

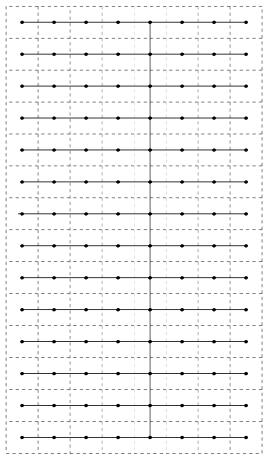

Figure 1.1: 14×8 **Row Pattern**, R

Note that we subtracted 1 from the distance in the tree to get d_T because we assume that pairwise stitching is free of error. We address the following optimization problems.

1. Find a spanning tree T of G that minimizes $S(T)$, the sum of the distances of the tree.
2. Find a spanning tree T of G that minimizes $M(T)$, the maximum of the distances of the tree.
3. Find a spanning tree T of G that minimizes $S(T)$, the sum of the distances of the tree, subject to $M(T)$ being as small as possible.

We calculate the total distance and the maximum distance for the row pattern R (Figure 1.1) of an $m \times n$ grid. It is optimal to centre the "spine" in the long direction. In this case $M(R) = 2\lfloor \frac{n}{2} \rfloor$ is as small as possible, and the total sum is given by:

$$S(R) = \begin{cases} (m-1)(n^2)/2 & \text{if } n \text{ is even} \\ (m-1)(n-1)(n+1)/2 & \text{if } n \text{ is odd} \end{cases} .$$

We note that this is the method currently employed.

1.2.2 Alternative Stitching Patterns

While the row pattern minimizes the maximum distance it does not minimize the total distance. In fact, one can replace a single edge of the tree R with a nearby edge and decrease the total sum. A pattern that reduces the total sum is the **Comb Pattern**, shown in Figure 1.2. This pattern is similar to the row pattern in that it has a long "spine", but it differs in that instead of long "lines" attached to the spine, it has "combs". It is again optimal in terms of $S(C)$ and $M(C)$ to centre the "spine" in the longest direction, as above. Then the maximum is $M(C) = 2\lfloor \frac{n}{2} \rfloor + 4 > M(R)$, but $S(C) < S(R)$. The values of $S(C)$ depend on the modulus of the parameters m and n, mod 3 and mod 2, respectively. Thus, there are six cases to consider; they are given in Table 1.1.

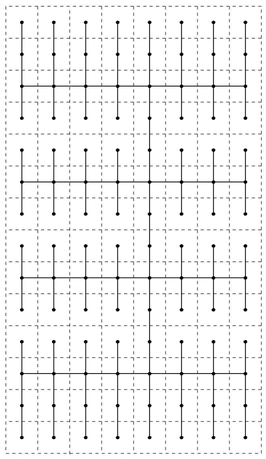

Figure 1.2: 14 × 8 Comb Pattern, C

m mod 3	n odd	n even
0	$\frac{1}{6}(m-3)(n-1)(n+17)+4(n-1)$	$\frac{1}{6}(m-3)(n^2+16n-16)+4(n-1)$
1	$\frac{1}{6}(m-4)(n-1)(n+17)+8(n-1)$	$\frac{1}{6}(m-4)(n^2+16n-16)+8(n-1)$
2	$\frac{1}{6}(m-5)(n-1)(n+17)+12(n-1)$	$\frac{1}{6}(m-5)(n^2+16n-16)+12(n-1)$

Table 1.1: Values for $S(C)$

The next method is to create a breadth-first search spanning tree which leads to what we call the **Breadth Pattern**, B. Let G_e be an $n \times n$ grid graph with n even and G_o be an $n \times (n-1)$ grid graph with n odd. Define an "almost square" grid graph to be a graph of the form G_e or G_o. We first define the breadth patterns B_e and B_o for these graphs. Label each vertex of the graph by the shortest distance to the perimeter of the grid. Figure 1.3 illustrates the 9 × 8 case. For the graph G_e, there are 6 vertices labeled $(n-3)/2$ in the centre and for the graph G_o, there are 4 vertices labeled $(n-2)/2$ in the centre. Begin with a spanning tree of these centre vertices as indicated in Figure 1.3.

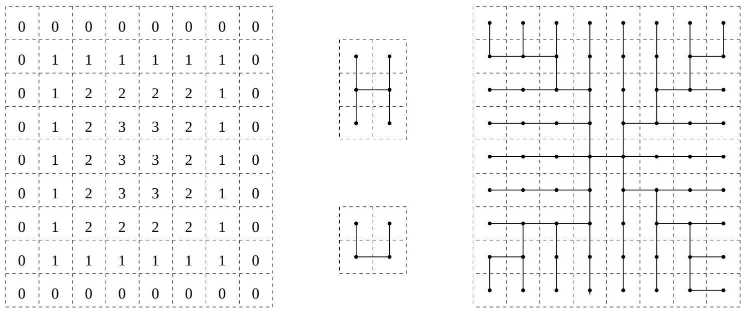

Figure 1.3: Breadth Pattern Construction

The procedure for growing the tree begins in the centre and radiates outwards. Suppose the tree has been grown to include all vertices labeled at least j, where $j > 0$. Add vertices labeled

$j-1$ as described below.

- If a vertex x with label $j-1$ is adjacent to a vertex y with label j, then y is unique. Add xy to the tree.
- If a vertex x with label $j-1$ is **not** adjacent to a vertex with label j, then x has two neighbors labeled $j-1$. Arbitrary choose one of these, say y, and add xy to the tree.

Figure 1.3 shows one possible breadth pattern for a 9×8 grid graph. We can easily calculate the sum of the distances for B_o and B_e in the almost square cases.

$$\begin{aligned} S(B_o) &= (n-1)(n+1)(2n-3)/6, \\ S(B_e) &= n(n+1)(n-1)/3. \end{aligned}$$

The breadth pattern, B, for any $m \times n$ grid graph is formed by placing half of the breadth pattern for the almost square grid on the top and bottom, and a row pattern in the middle. See Figure 1.4 for an example of this construction.

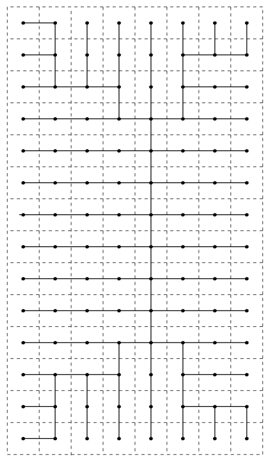

Figure 1.4: 14×8 Breadth Pattern, B

The maximum distance $M(B) = 2\lfloor\frac{n}{2}\rfloor$ is minimal, while the total sum is smaller than the row pattern sum. The values of $S(B)$ for the breadth pattern B of an $m \times n$ grid are given below.

$$S(B) = \begin{cases} (3mn^2 - n^3 - 2n)/6 & \text{if } n \text{ is even,} \\ (3mn^2 - n^3 - 3m + n)/6 & \text{if } n \text{ is odd.} \end{cases}$$

Comparing these patterns with the current row pattern, we see that the comb pattern improves the total sum at the expense of an increase in the maximum distance while the the breadth pattern improves the total sum while keeping the maximum distance at a minimum. We call a combination of these patterns a **Combed–Breadth Pattern** (Figure 1.5).

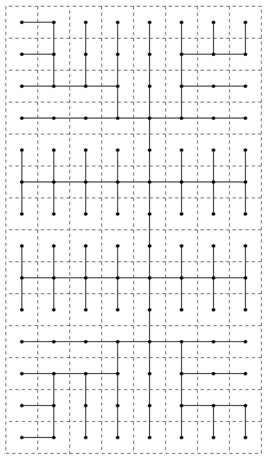

Figure 1.5: 14 × 8 Combed–Breadth Pattern, CB

1.2.3 Reliability

Not all the pairwise optimal offset values have the same confidence levels. We incorporate uncertainties by not allowing adjacent images to be stitched together if the corresponding offset values are highly uncertain. This is implemented by disallowing certain edges from the spanning tree.

This rule for excluding edges in the tree can be incorporated into breadth pattern. In Figure 1.6 the excluded edges are marked by grey barriers. Barriers were put where the uncertainty was higher than 7%. The tree is grown subject to the extra condition that it cannot grow across a barrier. We call this a **Barred Pattern**. One can also take the row pattern and the comb pattern and adjust them locally such that none of the edges of the spanning trees cross a barrier.

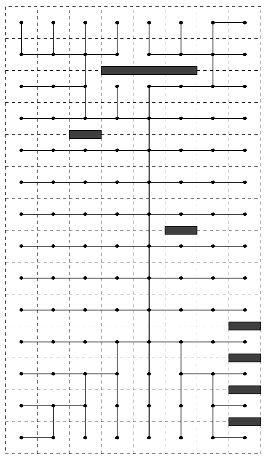

Figure 1.6: Barred Pattern, BA

In all cases, the numbers M and S cannot decrease. For this particular barred pattern, $M = 10$ for the modified row pattern and $M = 12$ for the modified comb and breadth patterns.

pattern	$S(\cdot)$	$M(\cdot)$
R (row)	416	8
C (comb)	234	12
B (breadth)	360	8
CB (combed-breadth)	258	12

Table 1.2: $S(\cdot)$ and $M(\cdot)$ values for a 14 × 8 grid

Also, the sum, S, increases by 4 for all three patterns. Table 1.2 contains calculations for Data set 1. In addition to the barred pattern in Figure 1.6 we also constructed a **Greedy Pattern** G, where we chose edges solely on the basis of confidence. The calculations for these patterns are also included in Table 1.2.

1.2.4 Data

Table 1.2 gives the $S(\cdot)$ and $M(\cdot)$ values for a 14×8 grid for the four tree patterns that we have defined.

Using the two sets of real data available, we evaluated the performance of each of these tree patterns. After stitching together the full image according to each pattern, we calculated the new pairwise relative offsets that each of these methods gave us. Then we calculated the differences both horizontally and vertically between these and the given "ideal" relative offsets. The larger a difference is, the more likely that it will create a significant misalignment that will lead to a problem in recreating the circuit. Thus we created three measures of the "badness" of each of our results.

The first measure was a sum of the absolute values of all of the horizontal and vertical differences between the relative offsets we obtained and the ideal relative offsets. This is denoted ABS ERROR. The second measure was chosen to highlight the fact that if a particular pair of images is placed (relative to one another) very far from their ideal relative alignment, this is worse than having a lot of images that are pairwise only slightly off their ideal alignment. Here we took the sum of the squares of all of the horizontal and vertical differences. This is labelled SQ. ERROR. The third measure took the threshold value of plus or minus 5 pixels as the worst tolerable error, and counted the number of values (horizontal or vertical) that were more than 5 pixels from their ideal relative alignment. This is labeled BAD VALUES in Tables 1.3, 1.4.

The barred and greedy patterns are present in Table 1.3 but not Table 1.4, since confidence data were only available for the first data set.

From this data, it appears that the breadth pattern produces the best results, although the barred pattern may improve on it in some ways. Similar to the barred pattern, one can modify the row and comb patterns to disallow certain edges. It seems that these highly regular patterns are quite sensitive to the precise locations of the barriers. If one changes an arbitrarily chosen edge in the spanning tree and replaces it in an optimal way, the breadth pattern performs much better than the row and comb patterns. This may be in part because there are many choices for constructing such a breadth pattern.

The greedy tree, although using high confidence values, resulted in extremely large values

pattern	ABS ERROR	SQ. ERROR	BAD VALUES
R	518	5092	32
C	513	5067	37
B	449	4229	30
CB	513	4897	39
BA	453	4309	27
G	721	7059	52

Table 1.3: Data Set 1

pattern	ABS ERROR	SQ. ERROR	BAD VALUES
R	569	6007	27
C	636	7560	34
B	517	5639	27
CB	619	7507	34

Table 1.4: Data Set 2

of M and S, and the effect of these is apparent in its poor performance. Of course, additional experimental data would be desirable, and it is likely that situations would arise in which different patterns provided the best performance, so one reasonable approach would be to maintain this as a small collection of trees to try on any given data set, and keep whichever final result is best. Some of these patterns could ultimately be discarded if additional experimental data showed them to consistently perform poorly.

We note that the breadth pattern continually outperforms the row pattern which is currently being used. In addition, the barred pattern may prove to be useful at a higher tolerance. Currently, a tolerance of 10% is used; however we chose 7% rather arbitrarily based on the one data set available. We also note that while the comb pattern has the minimum value of the total distances $S(C)$ it performs as well as the row pattern. Thus, it seems that the solution we seek will be the solution to optimization problem 3. Currently, the breadth pattern is the best candidate for a solution to that problem.

1.3 Least Squares

The pairwise offsets overdetermine the positions of images in the final grid. In this section we develop stitching solutions by simultaneously minimising the sum of the squares of the offset discrepancies.

1.3.1 Offset Equations

Let $(X_{i,j}, Y_{i,j})$ be the coordinates (in a global reference frame) for the northwest corner of the image in row i and column j (Figure 1.7). The grid of images has m rows and n columns.

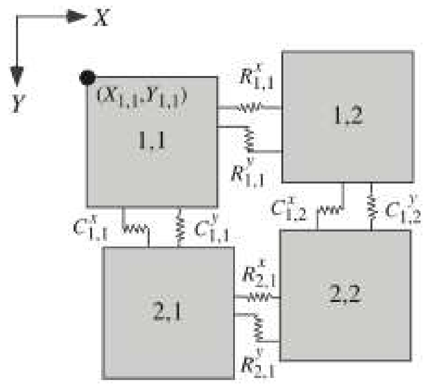

Figure 1.7: Coordinates and labelling of images in the grid. In least squares stitching the images can be interpreted as rigid plates connected by springs.

The x-component offsets $R^x_{i,j}$ between adjacent images in a row are described by the equations

$$X_{i,j+1} - X_{i,j} = R^x_{i,j} \quad (i = 1 : m,\ j = 1 : n-1)$$

These $m(n-1)$ row stitching equations can be written compactly as

$$XB_n^T = R^x \tag{1.1}$$

where B_n is the $(n-1) \times n$ bidiagonal matrix

$$B_n = \begin{bmatrix} -1 & 1 & & & \\ & -1 & 1 & & \\ & & \ddots & \ddots & \\ & & & & \\ & & & -1 & 1 \end{bmatrix}$$

The x-component offsets $C^x_{i,j}$ between adjacent images in a column are described by the equations

$$X_{i+1,j} - X_{i,j} = C^x_{i,j} \quad (i = 1 : m-1,\ j = 1 : n)$$

These $(m-1)n$ column stitching equations can be summarised as

$$B_m X = C^x \tag{1.2}$$

The foregoing discussion can be repeated for y-components, leading to the stitching equations

$$YB_n^T = R^y, \quad B_m Y = C^y$$

These equations are of the same form as the equations (1.1–1.2) for the x-components, and so may be treated in the same way. The y-component equations are not coupled with the x-component equations, so they may be computed separately.

In order to use standard software, it is convenient to convert the stitching equations into matrix-vector form. The Kronecker product is a standard technique for this sort of conversion [1]. The matrix of image location x-components X is converted into a vector by stacking the columns,

$$\bar{x} = \operatorname{vec}(X) = \begin{bmatrix} X_{1:m,1} \\ \vdots \\ X_{1:m,n} \end{bmatrix}$$

Converting the row stitching equations (1.1) gives

$$(B_n \otimes I_m)\bar{x} = \operatorname{vec}(R^x)$$

where I_m is the $m \times m$ identity matrix and $\otimes$ is the Kronecker product. Similarly, the column stitching equations (1.2) are converted to

$$(I_n \otimes B_m)\bar{x} = \operatorname{vec}(C^x)$$

The system of stitching equations is thus $\bar{A}\bar{x} = b$ where

$$\bar{A} = \begin{bmatrix} B_n \otimes I_m \\ I_n \otimes B_m \end{bmatrix}, \quad b = \begin{bmatrix} \operatorname{vec}(R^x) \\ \operatorname{vec}(C^x) \end{bmatrix}$$

Finally, we impose the condition $\bar{x}_1 = X_{1,1} = 0$, which fixes the global coordinate system's origin. This yields the system

$$Ax = b \tag{1.3}$$

where A is $\bar{A}$ with the first column deleted, and x is $\bar{x}$ with the first element deleted. The system (1.3) has full column rank.

1.3.2 Least squares

Equation system (1.3) has $(m-1)n + m(n-1)$ stitching equations for the $mn - 1$ unknowns in x. This system of equations is overdetermined whenever $m > 1$ and $n > 1$. Unless the offsets R^x and C^x happen to be consistent, any set X of image locations will give some nonzero error ("residual") in the equations.

One way to determine X is to seek the minimiser of the sum of squares of the residuals of equations (1.1–1.2) subject to the grounding condition $X_{1,1} = 0$. This is equivalent to the unconstrained minimisation of

$$(Ax - b)^T (Ax - b)$$

For a physical interpretation of the least squares formulation, imagine the grid as an array of rigid plates connected by elastic springs (Figure 1.7). A stitching equation that is satisfied corresponds to a spring that is neither stretched nor contracted. There are more springs than there are degrees of freedom, so the springs stretch or contract to find the configuration of minimum potential energy. This configuration is the least squares solution, and the residuals are the extensions in the springs.

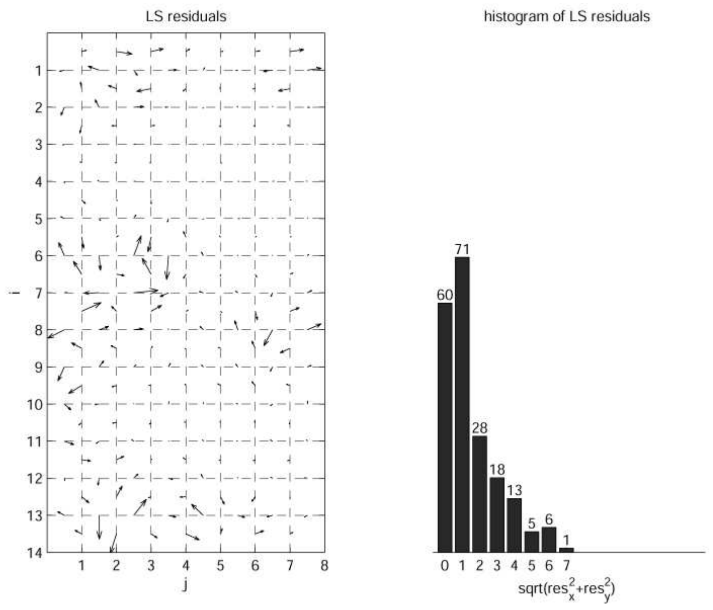

Figure 1.8: Residuals for least squares stitching

The least squares minimiser is the solution of the square system of "normal" equations

$$A^T Ax = A^T b$$

The normal equation system is nonsingular because A has full rank, and it can be solved reliably and efficiently using standard sparse linear algebra software.

The Matlab script `LSstitching.m` [2] does least squares stitching of a 14 by 8 grid of images using offset data provided by the problem proposer. Computing time on a 400 MHz Macintosh is 0.02 s. The least squares solution's residuals are mostly 0–3 pixels, although a few seams have residuals of 4–7 pixels (Figure 1.8). The problem proposer did a visual inspection of the least squares solution and reported that all but one of the seams appears to be OK.

1.3.3 Constrained Least Squares

In order to reduce the maximum residual obtained by the least squares solution we impose the inequality constraints

$$-\alpha \leq b - Ax \leq \alpha$$

where α is the desired limit on the residual's absolute value. The smallest value of α that gives a feasible solution can be found by trial and error or by solving a linear programming problem. The constrained least squares problem can be solved using standard optimisation software packages.

In script `CLSstitching.m` [2] we used the Matlab Optimization Toolbox (v. 1.5.2) routine `conls`. For the 14 by 8 grid the smallest feasible α was 4 pixels for the x-components and 5 pixels for the y-components. The maximum residual thus has length $\sqrt{4^2 + 5^2} \approx 6$, which is only 1 pixel smaller than for the unconstrained least squares solution, and many more of the residuals are large (Figure 2), so this solution seems to be of little practical value.

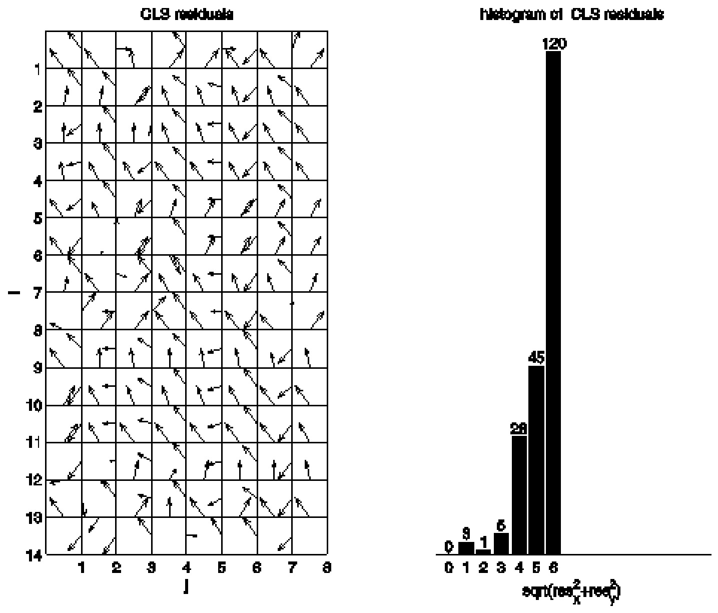

Figure 1.9: Residuals for constrained least squares stitching

1.3.4 Weighted Least Squares

Another approach to improving the practical quality of the stitching is to minimize the weighted sum of squares of the residuals,

$$(Ax - b)^T W^2 (Ax - b)$$

where the weighting matrix W is a nonsingular diagonal matrix. The diagonal elements of W can be interpreted as stiffness constants for the springs holding together the rigid plates in Figure 1.7. With $W = I$ all the springs have equal stiffness, and we revert to the original "unweighted" least squares problem. The weighted least squares problem can be solved by simply replacing A and b in the unweighted problem by WA and Wb.

One possible strategy for assigning the weights would be to give larger weights to those equations where the quality of the stitching is most sensitive to stretching or shear. For example, in parts of the images that contain a lot of parallel lines we want the lines to match up correctly at a seam, so we assign a large stiffness to the appropriate spring. A seam that contains no features would be assigned a small stiffness. It might be possible to derive this kind of sensitivity information automatically from the correlation data that were used by the problem proposer to compute the offsets; we would need more information about the images and the correlations to explore this possibility.

Another strategy would be to assign the weights using interactive graphics. Based on the solution of the unweighted least squares problem, a human operator would examine the seams where stitching quality is unacceptable, adjust those seams' stretching and shearing stiffnesses, and recompute the weighted least squares solution. This process could be iterated until (hopefully!) all the seams were acceptable. If the initial solution gives only a small number of unacceptable seams, this interactive adjustment would not be too laborious.

A third strategy, which uses only the information given in the problem, is to assign small

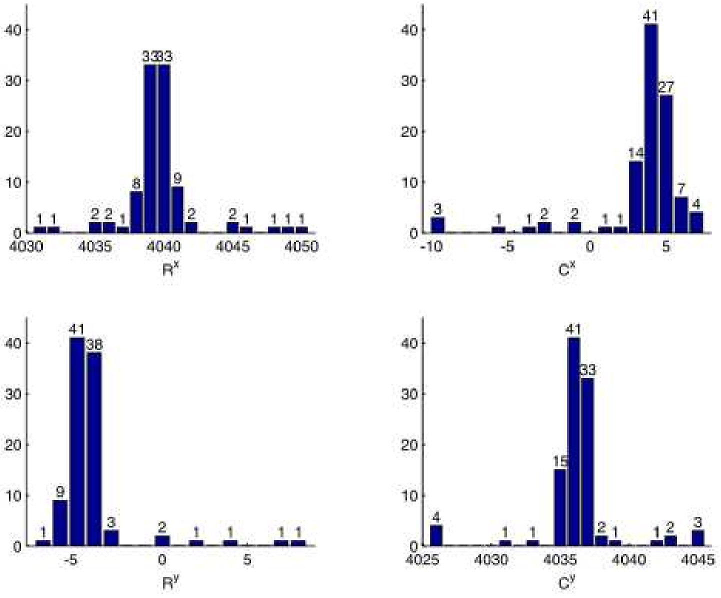

Figure 1.10: Histograms of offset values

weights to those equations whose offset value differs significantly from the mean value. The rationale is that these offsets are erroneous ("outliers"), because of some sort of measurement error, and should be ignored.

Applying this third strategy to the 14 by 8 grid, we assigned weights of $\sqrt{0.1}$ to those equations whose offset value differed by more than 3 pixels from the mean (Figure 1.10); the remaining equations' weights were 1.0. In the weighted least squares solution (`WLSstitching.m` [2]) the residuals of the outliers tends to increase (compared to their values in the unweighted least squares solution), but the remaining residuals mostly decrease (Figure 1.11). The problem proposer reports that with the weighted least squares solution all the seams appear to be OK, so this strategy seems to be promising.

1.3.5 Least Squares with Deformable Images

A way of reducing the residual at all the seams is to introduce more degrees of freedom by allowing the images to deform in various ways. We introduced deformability by dividing each image into four equal subimages. Theoretically, the sum of squares of the residuals scaled by the length of the seams of the subimages should decrease. Applying this approach to the 8 by 14 grid gave a maximum residual of 5 pixels, which is 2 pixels less than with the original least squares solution. We believe that more sophisticated deformation models, combined with weighting and more offset data (for example, measuring several x-offsets along an east-west seam), could bring all the seam residuals down to less than 1 pixel.

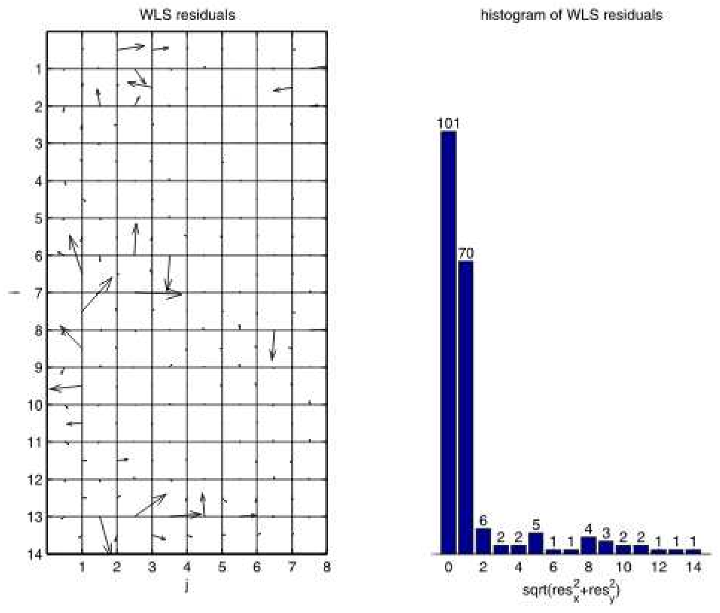

Figure 1.11: Residuals for weighted least squares stitching

1.4 Simulated Annealing

A more fundamental approach to stitching is to directly minimise an error function computed from the overlapping pixels in the entire image. In this section the minimisation problem is solved using the method of simulated annealing. Continuous minimisation methods for this problem are studied in section 1.5.

Simulated annealing is one of a family of meta-heuristics that are able to solve instances of hard optimization problems. These meta-heuristics take advantage of the ability to quickly define a local perturbation and then to recalculate the global objective function by a fast, local calculation. This family of meta-heuristics also includes *hill-climbing, Tabu search, genetic algorithms,* and *rising flood* techniques. The central problem is to avoid local extrema. Simulated annealing does this by analogy to cooling solids. Initially the setting is "hot", meaning that perturbations that worsen the objective function are sometimes allowed. We gradually "cool down" the setting, meaning that fewer bad moves are acceptable, as we hone in on better objective values. To avoid getting stuck in a local extremum, we periodically crank up the heat and again slowly cool. Repeating this leads to surprisingly good solutions.

We wrote a C++ simulated annealing program for image stitching. It is licensed under the GNU GPL and available for download [3]. The program has sophisticated error handling routines that identify well-formatted input data. Input parameters include minimum feature size, image translation bounds, temperature, number of reheating and recooling stages, and temperature thresholds.

We first defined an objective function that measured how far each image is placed from its ideal offsets using an error threshold defined by the minimum feature size. A seam whose error is less than this threshold in both coordinates contributes 0 to the objective function, otherwise it contributes 1. The program randomly selects an image to move and makes a small translation of the image, locally recalculating the overall objective function by comparison to the ideal offsets for that single image. This objective function always rapidly diverged to the worst case (all boundaries bad) and was never able to improve. We believe the problem was the fact that the objective function had no way to recognize if a boundary was non-optimal but very close. It registers the same value as the boundary being far from a good fit. Thus once a boundary was bad, the vast majority of perturbations would leave the objective function unchanged leading to wandering on plateau in the state space.

We then defined an objective function that scored 0 if the alignment is within a feature size and scored a value that was proportional to how far away from correct the boundary was if it was not a good fit. This rapidly converges to a good layout.

Instead of doing merely local translations, it is straightforward to incorporate small rotational changes, shrinking and expanding changes, and slight skewing changes into this simulated annealing algorithm. This would require subroutines to rapidly recalculate alignment scores along changed borders between images. After performing the above generalized perturbations on a single image, we would compute the correlations of pairs of images to determine how the changed image correlates with adjacent images, and use that to measure the fitness of the image. If these routines are quick then the algorithm could address the very real concern that stack movement and other noise may produce skewed, scaled, or rotated images.

1.5 Nonlinear Programming

1.5.1 Cost Function

In this section we represent each image as a matrix $[u^{ij}]_{L_y \times L_x}$ of integers (corresponding to pixel grey scale values). There are target offset overlaps (S_x and S_y) which provide overlap regions to help align the images. Assuming that the target overlaps are achieved, the top left corner of each image define the reference grid (Figure 1.12). In practice however, errors during image acquisition modify these overlap regions. This can be quantified by associating an offset error $\vec{\Delta}^{pq} = (\Delta_x^{pq}, \Delta_y^{pq})$ with every image u^{pq}. The offset error is defined relative to the reference grid. The left overlap region between image u^{ij} and $u^{i,j-1}$ has width $\big(S_x - (\Delta_x^{ij} - \Delta_x^{i,j-1})\big)$ and height $\big(L_y - (\Delta_y^{ij} - \Delta_y^{i,j-1})\big)$. Similarly, the top overlap region between image u^{ij} and $u^{i-1,j}$ has width $\big(L_x - (\Delta_x^{ij} - \Delta_x^{i-1,j})\big)$ and height $\big(S_y - (\Delta_y^{ij} - \Delta_y^{i-1,j})\big)$.

It is also assumed that we have an upper bound on the offset errors, i.e., $|\Delta_x^{ij}| \leq \beta_x$ and $|\Delta_y^{ij}| \leq \beta_y$ where β_x and β_y are constants, or, an upper bound on the relative offset errors between every pair of neighbouring images. Let

$$\eta_x = \Delta_x^{i,j-1} - \Delta_x^{ij}, \tag{1.4a}$$

$$\eta_y = \Delta_y^{i,j-1} - \Delta_y^{ij}, \tag{1.4b}$$

$$\xi_x = \Delta_x^{i-1,j} - \Delta_x^{ij}, \tag{1.4c}$$

$$\xi_y = \Delta_y^{i-1,j} - \Delta_y^{ij}, \tag{1.4d}$$

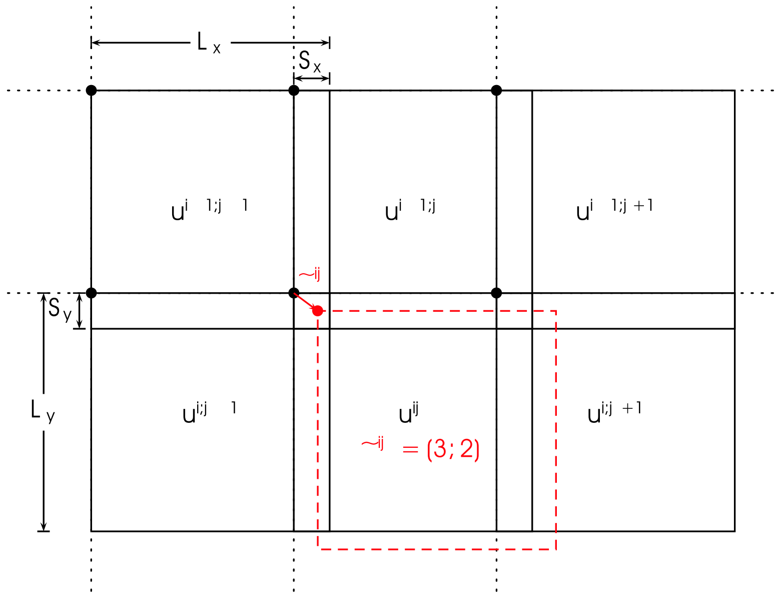

Figure 1.12: Image layout using reference grid approach. Solid lines show images at target offset overlaps (upper left corners are shown as ●), dashed box shows image u^{ij} displaced by offset error $\vec{\Delta}^{ij}$.

and thus

$$\begin{aligned} |\eta_x| \le \beta_x, \quad |\eta_y| \le \beta_y, \\ |\xi_x| \le \beta_x, \quad |\xi_y| \le \beta_y. \end{aligned}$$

It is unclear from the problem specification whether the β_x and β_y bound the relative offset errors or the actual offset errors, but the following discussion and code handles both cases.

We calculate the cost function by summing the contributions from the top and left overlap regions for all images u^{ij}. The cost function contribution due to u^{ij} is

$$\begin{aligned} d^{ij} &= \frac{\sum\sum_{\text{left overlap pixels}} \left|u^{ij} - u^{i,j-1}\right|^2}{\#\text{ of left overlap pixels}} + \frac{\sum\sum_{\text{top overlap pixels}} \left|u^{ij} - u^{i-1,j}\right|^2}{\#\text{ of top overlap pixels}}, \\ &= \text{cost\_contrib\_left}(u^{ij},\, u^{i,j-1},\, \eta_x, \eta_y) \\ &\qquad + \text{cost\_contrib\_top}(u^{ij},\, u^{i-1,j},\, \xi_x, \xi_y), \end{aligned} \tag{1.5}$$

where the first term is 0 if $j = 1$ and the second term is 0 if $i = 1$. Note that d^{ij} depends explicitly on $\vec{\Delta}^{ij}$, $\vec{\Delta}^{i-1,j}$ and $\vec{\Delta}^{i,j-1}$. The total cost function is then calculated as

$$d = \sum_i \sum_j d^{ij}. \tag{1.6}$$

1.5.2 Minimization

At this point, we try minimizing the above cost function using Matlab's *fminsearch* over the space of $\vec{\Delta}$'s. However, we encounter an unexpected hurdle: the stitching of images is not a discrete problem whereas our cost function calculation assumes that the offset errors were integers. One may question the usefulness of resolving the offset errors to a "sub-pixel" level when feature sizes are $\mathcal{O}(10)$ pixels, however, finding a free black-box optimization code which generates and *uses* only integer values is difficult. We propose several fixes:

- round $\vec{\Delta}^{ij}$ when calculating the cost function.
- interpolate the image data on the sub-pixel level.
- interpolate our cost function.

As expected, rounding does not produce a convergent result. Interpolating the image is impractical for larger problems where realistically, a caching approach as discussed in Section 1.5.2 is required. We choose to implement interpolation of the cost function.

Cost Function Interpolation

Let cost_contrib_left() be a function that calculates the cost function contribution in the left overlap region between neighbouring images u^{ij} and $u^{i,j-1}$. Assuming that η_x and η_y are integers, the cost contribution is calculated as

$$\text{cost\_contrib\_left}(u^{ij},\, u^{i,j-1},\, \eta_x, \eta_y).$$

To accommodate fractional offset errors, consider

$$\eta_x = \lfloor \eta_x \rfloor + \delta_x, \tag{1.7a}$$
$$\eta_y = \lfloor \eta_y \rfloor + \delta_y, \tag{1.7b}$$

where $\vec{\delta} \in [0,1) \times [0,1)$, $\lfloor \cdot \rfloor = \text{floor}(\cdot)$ and $\lceil \cdot \rceil = \text{ceil}(\cdot)$.

We now build the lookup table for the cost function contribution at the 4 nearest neighbouring integer values of $\vec{\eta}$ by calculating

$$c_{\lfloor x \rfloor \lceil y \rceil} = \text{cost\_contrib\_left}(u^{ij},\, u^{i,j-1},\, \lfloor \eta_x \rfloor, \lceil \eta_y \rceil), \tag{1.8a}$$
$$c_{\lceil x \rceil \lceil y \rceil} = \text{cost\_contrib\_left}(u^{ij},\, u^{i,j-1},\, \lceil \eta_x \rceil, \lceil \eta_y \rceil), \tag{1.8b}$$
$$c_{\lfloor x \rfloor \lfloor y \rfloor} = \text{cost\_contrib\_left}(u^{ij},\, u^{i,j-1},\, \lfloor \eta_x \rfloor, \lfloor \eta_y \rfloor), \tag{1.8c}$$
$$c_{\lceil x \rceil \lfloor y \rfloor} = \text{cost\_contrib\_left}(u^{ij},\, u^{i,j-1},\, \lceil \eta_x \rceil, \lfloor \eta_y \rfloor). \tag{1.8d}$$

Our estimate, c of the cost function contribution is a weighted average of (1.8). Specifically, we use bilinear interpolation, by first calculating

$$c_{\lceil x \rceil} = (1 - \delta_y) c_{\lceil x \rceil \lfloor y \rfloor} + \delta_y c_{\lceil x \rceil \lceil y \rceil},$$
$$c_{\lfloor x \rfloor} = (1 - \delta_y) c_{\lfloor x \rfloor \lfloor y \rfloor} + \delta_y c_{\lfloor x \rfloor \lceil y \rceil},$$

and then calculating

$$c = (1 - \delta_x) c_{\lfloor x \rfloor} + \delta_x c_{\lceil x \rceil}.$$

A similar discussion applies for cost_contrib_up() using ξ_x and ξ_y.

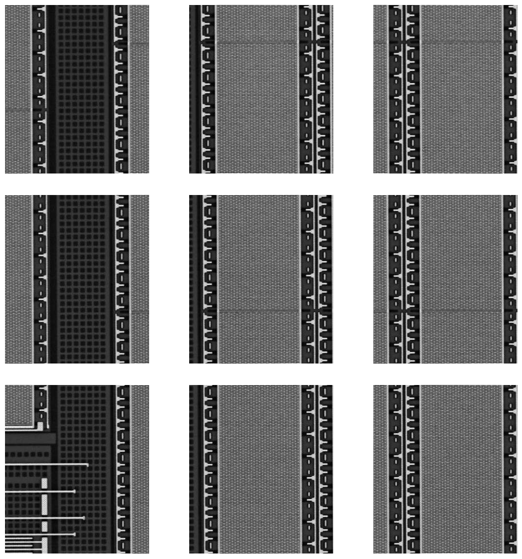

Figure 1.13: Test Case 1.

Cost Function Space

The set of images above was used as Test Case 1. The original image provided by Semiconductor Insights of a chip was artificially broken into nine 700×600 pieces with overlap regions of 100 pixels and random offset errors in the range of -6 to 6 in both the horizontal and vertical direction (Figure 1.13).

Unfortunately, given a random initial guess for the $\vec{\Delta}$'s, *fminsearch* could not find the global minimizer. We present two figures of the function space in question. Given two sets of offset errors say $\vec{\Delta}$ and $\vec{\Delta}_{\text{exact}}$, we define the radius r as $\|\vec{\Delta} - \vec{\Delta}_{\text{exact}}\|$. Figure 1.14 shows the cost as a function of radius for fixed directions $\hat{\theta}$ from the exact solution, i.e., define

$$\hat{\theta} = \frac{\vec{\Delta} - \vec{\Delta}_{exact}}{\|\vec{\Delta} - \vec{\Delta}_{exact}\|} \tag{1.9}$$

and plot cost as a function of r. Notice that none of curves are monotonic, in fact, a plot of cost as a function of radius for 1000 $\hat{\theta}$'s reveal that only one or two curves were monotone. The cost function space has a lot of local mimima; this presents problems for most optimization routines.

We also present the plots obtained by keeping most of the offset variables constant at the known exact errors while varying pairs of offset variables (Figure 1.15). One can clearly see the existence of many local minimas. Close to the exact solution however, the function spaces are very smooth (Figure 1.16). This provides some hope that an optimization scheme can indeed be used, provided a good initial guess is given.

Iterative Scheme

To provide a better initial guess for the optimization routine, we adopt an iterative approach to the problem. This involves stitching one additional image each iteration.

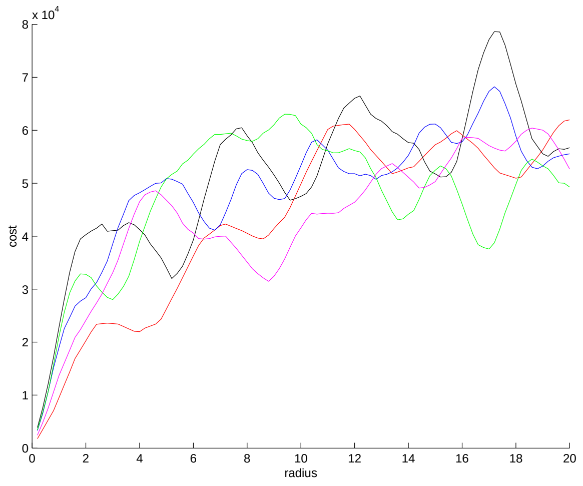

Figure 1.14: Cost as a function of radius for various directions in offset error space.

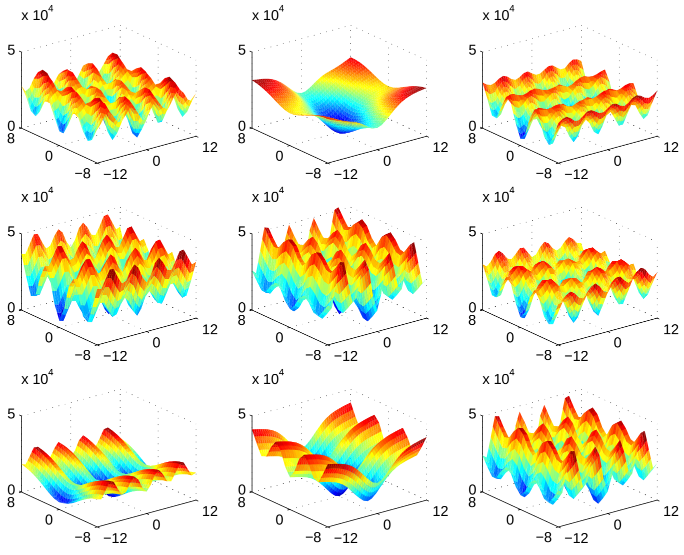

Figure 1.15: Multiple plots of the cost function space for arbitrary pairs of offset variables. (Other variables held at constant, correct minimizing values.)

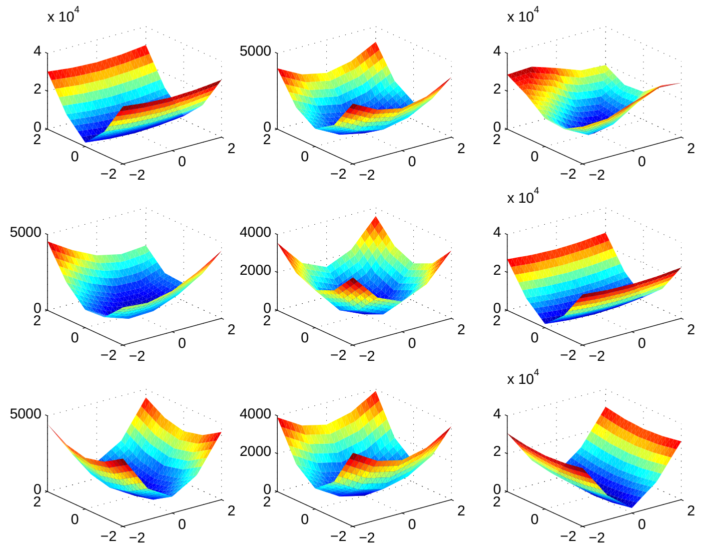

Figure 1.16: Multiple plots of the cost function space close to the global minimum. Also note the effect of bilinear interpolation.

We begin by choosing a particular ordering of the images, $(u^1, u^2, \ldots, u^K)$, $K = M * N$ where each u^k corresponds to a unique u^{ij} and u^k shares at least one common edge with $(u^1, \ldots, u^{k-1})$. A simple ordering is

$$\begin{matrix} u^1 & u^2 & u^3 \\ u^4 & u^5 & u^6 \end{matrix}$$

More effective orderings are considered in Section 1.2.

The following pseudo-algorithm then illustrates the process of sequentially stitching one additional image per iteration:

1. Assume that $\vec{\Delta}^1 \equiv \vec{0}$ and set $k = 2$.
2. Stitch $u^1, \ldots, u^k$ assuming that the known $\vec{\Delta}^1, \ldots, \vec{\Delta}^{k-1}$ are fixed and only $\vec{\Delta}^k$ is allowed to change, i.e., find the optimal $\vec{\Delta}^k$ which stiches the images. (Note, we use a random initial guess for $\vec{\Delta}^k$)
3. Stitch $u^1, \ldots, u^k$ varying $\vec{\Delta}^1, \ldots, \vec{\Delta}^k$, using the values from step 2 as the initial guess.
4. Set $k = k + 1$ and return to step 2.

We acheived correct convergent results in approximately 90% of the trials, however, the convergence was very slow. On a 3×3 set of images (i.e., 14 offset variables), the code took over eight hours to complete on a 1.2 GHz Athlon MP.

Pre-Conditioner

In retrospect, it is not necessary and quite inefficient to take a random initial guess for $\vec{\Delta}^k$ in step 2, Section 1.5.2. Figure 1.16 shows that an initial guess within two pixels is likely sufficient for convergence.

A better approach is to do an exhaustive search over a finite set of values for $\vec{\Delta}^k$, and use the $\vec{\Delta}^k$ providing the smallest contribution to the cost function as the initial guess in step 3 of Section 1.5.2. A natural space of relative offset errors is $\{\vec{\Delta} \in \mathbb{Z}, |\vec{\eta}| \leq |\vec{\beta}|\}$ (where $\vec{\beta} = (\beta_x, \beta_y)$) if we have bounds on the offset errors. If we instead have bounds on the relative error between images, we can do something similar: $\{\vec{\eta} \in \mathbb{Z}, |\vec{\eta}| \leq |\vec{\beta}|\}$ where the inequalities are between u^{ij} and $u^{i-1,j}$ unless $i \equiv 1$, in which case the inequalities are between u^{ij} and $u^{i,j-1}$. An important point to realize is that although our formulation uses a fixed reference grid, the contribution to the cost function between neighbouring images only depends on the relative difference in offset errors between them.

Upon implementation of the pre-conditioner, our code became more robust, with a noticable increase in speed. We were actually quite surprised at how robust the code was. It was unclear if our results were attributed to the artificial construction of test images (cutting a single picture into multiple parts), so we tested our code on a variety of altered images, for example:

- After cutting up the image into smaller test images, each test image was saved as a jpeg with a quality rating of 20. Quality ratings for jpeg range from 0 to 100 where a higher number means less image degradation due to compression.

- Each test image was also subjected to a filter which added gaussian bluring (within a radius of 5 pixels).

- Each test image was subjected to a noise filter. This noise filter "jittered" the image by randomly perturbing some pixels a small distance[1]. This filter actually made the images look quite horrible and hard to optimize.

- The output from the pre-conditioner was perturbed by $|\epsilon| < 1$ to simulate the case when the pre-conditioner does not find the correct values.

Our code still found the global minimum for all the test cases, giving us confidence that it could work when real images are used. Lots of nasty attributes such as lens abberation, difference in contrasts, warping of images would be noticed in real images.

Caching

A co-author suggested that caching the cost function contribution for various $\vec{\Delta}$'s would provide significant speedup. This should translate to significant computational savings during our bilinear interpolation. Another important motivation is the difficulty in storing a 8×14 grid of 8k $\times$ 8k images in memory (7 Gb at one byte per pixel). There are two possible stages to implement caching: pre-caching the cost function contributions before the optimization step or caching during the optimization step (i.e., cache the cost contribution for each new integer $\vec{\Delta}$).

[1] "pick" filter in GIMP (www.gimp.org)

The former approach was taken because there is less book-keeping involved. It is likely though that caching during the optimization step will provide further speedup.

For every image u^{ij}, associate two matrices D_L and D_U which correspond to cached contributions from the left and top image respectively. As mentioned in Section 1.5.2, although our formulation uses the fixed reference grid, the contribution to the cost function from neighbouring images depends only on the relative difference in offset errors. We utilize the experimental constraint that the offset errors are bounded above by $\vec{\beta}$. Thus, D_L and D_U are matrices with $(2\beta_y+1)\times(2\beta_x+1)$ elements. Note, in practice, we cache a larger matrix to account for roundoff error.

For a 6×5 grid of 700×600 images (Test Case 2) with $\beta_x = 10$ and $\beta_y = 8$, caching took 7 minutes and optimization took 14 minutes. Note, our non-caching code for this problem was halted after 4 days. A rough estimate for caching the 14×8 problem described above is approximately nine hours. Note that the caching algorithm scales linearly with the number of images and linearly with the size of the overlap region. If the computation time is too long, the caching algorithm is easily parallelizable for a cluster type network.

1.5.3 Results

The optimization code encounters no difficulties fitting Test Case 2. A rough extrapolation suggests that the 14×8 problem with 240 variables would take about one day.

Matlab codes and more details on this work are available online[4]. The authors of this Section thank Jim Verner for proof-reading and constructive criticism and Jason Nielsen for help with extrapolation techniques.

1.5.4 Future Work

- One should *actually try* the fixed reference grid approach of sequentially fitting images. Global optimization may not be necessary.

- Combine the iterative scheme in Section 1.5.2 with a more effective ordering considered in Section 1.2.

- Employ a strategy whereby offset variables from images that have enough surrounding neighbours are removed from the optimization process. One could use this process to bound the number of parameters in the optimization.

- Throughout this discussion, we have assumed that the images have been pre-processed, e.g., the images have been rotated correctly. The pre-processing stage can be incorporated into the cost function.

- Recode our algorithm in a more efficient environment such as C or Fortran.

- The cost function is easily modified to fit two different layers of images. A first step would be to detect via-ups and via-downs in each image/layer. (A good segmentation algorithm based on levelsets or wavelets would probably do the job.) One could then calculate the cost function for each layer, sum them together to create the new cost function, and add a penalty term to the new cost function if the vias connect to wires either above or below. The parameter space for two layers of 14×8 images can be handled by a desktop computer in a reasonable amount of time.

- This problem can be reformulated to incorporate the power of parallel processing. For example, the caching routine is easily parallelized.
- For large images, it is not necessary to read in the whole image when optimizing for the offset errors. In fact, less than 5% of the image is required assuming an overlap width of 100 on 8k $\times$ 8k pictures. Incorporating this idea will relax memory constraints.
- Custom tailor a minimizer that pays attention to the pixel scale. It can use sub-pixel information if necessary but ideally should not because sub-pixel information is not important. A first attempt would probably involve a simple algorithm:
    1. Calculate an approximation to the Jacobian (finite differences).
    2. Calculate an approximation to the Hessian (finite differences).
    3. Calculate a search direction based on the Jacobian and Hessian (simple line search).
    4. Choose an appropriate step in the search direction - i.e., integer values.
    5. Evaluating the Hessian and Jacobian is probably going to be quite expensive, so it's probably better to "Freeze" the Jacobian and Hessian by returning to step 3 and recalculating a search direction.
- If the images were collected in a hexagonal pattern (i.e., acquisition of images for every other row is offset by $\frac{L_x}{2}$) then one would expect a better alignment due to increased connectivity.
- A more radical approach would be to incorporate (local) stitching into the circuit generation code. The code would follow traces, and whenever it encountered the edge of an image, it would simply do a local stitching with the neighbouring image and continue the trace onto that image.

1.6 Conclusions

The different solutions presented here require different levels of effort to be implemented, with corresponding different potential payoff. The graph theory solutions of Section 1.2 provide improvements on the patterns of sequential stitching currently used. The least squares codes of Section 1.3 fairly rapidly compute stitching solutions based on the offset data. The simulated annealing and nonlinear programming codes of Sections 1.4–1.5 are more flexible, because they can find optimal stitchings based on the image pixel data, but they are computationally much more demanding.

Bibliography

[1] James W. Demmel, *Applied Numerical Linear Algebra*, SIAM, 1997, Section 6.3.3.

[2] `ftp://ftp.cc.tut.fi/pub/math/piche/stitching/`

[3] `http://mathstat.carleton.ca/~brett/Research/Code/#ipsw02`

[4] `http://www.math.sfu.ca/~bwo/ipsw`