MIST Algorithm Documentation
(Microscopy Image Stitching Tool)

Michael Majurski, Timothy Blattner, Joe Chalfoun, Walid Keyrouz
NIST/ITL/SSD
{michael.majurski, timothy.blattner, joe.chalfoun, walid.keyrouz}@nist.gov

Version 2018-05-21

Contents

1 Introduction

This document outlines the MIST algorithm. It describes the particular variant of the Image Stitching problem being addressed and outlines the current solution strategy. The algorithm consists of three phases: (1) relative

translation computation; (2) relative translation optimization; (3) image composition. Each phase is presented in a depth first hierarchy.

The document will be treated as a live document that will be updated regularly to always reflect the group's current thinking about the project.

1.1 Problem Description

In our context, the Image Stitching problem comes up when an optical microscope, equipped with a digital camera, generates overlapping partial images (*a.k.a.* image tiles) of a biological sample (plate) under study. We must then use specialized software ("Image Stitching" software) to assemble these image tiles into a single large image. We further restrict the problem by specifying that the image tiles are *grayscale* images where each pixel contains a single scalar value.

This problem arises from a scale mismatch between the dimensions of the sample being studied and the microscope's field of view. Typically, the region of interest in a microscope's plate is a $2 \times 2\,\mathrm{cm}^2$ area ($20 \times 20\,\mathrm{mm}^2$). By contrast, a microscope's field of view has sides that are at least one order of magnitude smaller ($\approx 1 \times 1\,\mathrm{mm}^2$). For a particular microscope currently used at NIST, a partial image is 1392×1040 pixels with each pixel covering a square area whose side is $0.644 \mu m$.

To alleviate the scale mismatch, the sample is mounted on a motorized stage and moves with respect to the optical column in the XY-plane. The microscope is then programmed to scan the XY-plane and repeatedly acquire image tiles, partial pictures of the plate, to form an image grid. The image tiles are arranged in a rectangular grid and overlap with each other to provide full coverage of the plate. We further restrict the problem by specifying that the overlaps between images must be approximately constant in the horizontal direction and approximately constant in the vertical direction. Figure 1 shows a demonstration image grid consisting of 25 image tiles with low overlap.

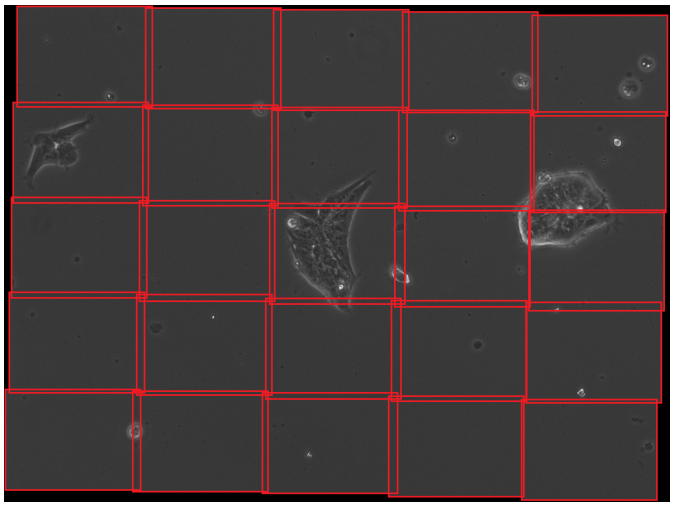

Figure 1: Example 5x5 Image Grid with individual images outlined in red.

1.2 Algorithm Overview

1.2.1 Translation Computation (Algorithm 1) Method Call Hierarchy

1. Phase Correlation Image Alignment Method (Algorithm 2)
   (a) Peak Correlation Matrix (Algorithm 3)
   (b) Multi-Peak Max (Algorithm 4)
   (c) Interpret Translations (Algorithm 5)

i. Extract Overlap Subregion (Algorithm 7)

ii. Normalized Cross Correlation (Algorithm 6)

1.2.2 Translation Optimization (Algorithm 8) Method Call Hierarchy:

Translation Optimization (Algorithm 8) Method Call Hierarchy:

1. Build Stage Model (Algorithm 9)
   (a) Compute Image Overlap (Algorithm 10)
   (b) Compute Stage Repeatability (Algorithm 13)
2. Apply Stage Model (Algorithm 14)
   (a) Filter by Overlap and Correlation (Algorithm 15)
   (b) Filter by Outliers (Algorithm 16)
   (c) Filter by Repeatability (Algorithm 17)
   (d) Replace Invalid Translations with Estimates
      i. Replace Invalid Translations (Algorithm 18)
      ii. Estimate Empty Row/Columns (Algorithm 19)
3. Bounded Translation Refinement (*ncc* hill climbing) (Algorithm 21)

1.3 Terminology

This section introduces terminology used throughout this document.

- Image Grid: a two dimensional array where each element contains a single image. The image grid is acquired by the microscope using a motorized XY-Stage. It can be thought of as the input to this stitching algorithm. An image's position within this grid represents an approximation of where that image should be placed relative to the other images in the grid, $imgGrid[0,0]$ (the top left array element) will be placed in the upper left hand corner of the output mosaic image. Likewise, $imgGrid[height-1, width-1]$ (bottom right array element) will be placed in the bottom right hand corner of the output mosaic image. Empty elements (*a.k.a.* missing images) are permissible so long as the set of non-empty elements forms a single connected graph.
- Image: a two dimensional array of *grayscale* pixel values. The x coordinate of a pixel increases from left to right and the y coordinate of a pixel increases from top to bottom.
- Image Tile: an image within the image grid. Each image tile has three ancillary attributes: (1) an (x, y) location in the output stitched mosaic image; (2) a North translation tuple containing the displacement between this image and its North neighbor; and (3) a West translation tuple containing the displacement between this image and its West neighbor.
- Translation: a triplet, $\langle ncc, x, y\rangle$, specifying the translation between two adjacent image tiles. The *ncc* value represents translation quality; x is the horizontal translation in pixels; y is the vertical translation in pixels.
- Tuple: a set of two or more numbers which, when grouped together, form a single concept.
- PCIAM: (phase correlation image alignment method) an algorithm to align (register) a pair of equally sized overlapping images. This method generates a translation tuple $\langle ncc, x, y\rangle$ denoting the displacement from one image to the other [3, 4, 5].
- *ncc*: (normalized cross correlation) a double precision value $\in [-1, 1]$ that represents a measure of similarity of the overlapping subregion between two images given a displacement (x, y) from one image to the other.

1.4 Organization

The remainder of this document is organized as follows: Section 2 describes the motorized XY-Stage movement model and the assumptions therein; Section 3 describes the translation computation phase giving a pseudo-code listing of it; Section 4 describes the translation optimization phase giving a pseudo-code listing of it; Section 5 briefly describes the image composition phase; Section **??** describes the algorithm testing and validation.

2 XY-Stage Actuator Movement Model

A motorized mechanical XY-Stage is used to move a biological sample with respect to the microscope's optical imaging system. This movement is done by two independent stepper motor linear actuators, one for each direction.

It is important to understand the mechanical limitations of a stepper motor linear actuator mainly with regards to its accuracy and repeatability.

Accuracy: Actuator motion can only be as true as the mechanical components permit. The accuracy is the actuator's ability to provide precise increments of motion along its axis. In a mechanical system, this primarily concerns the lead screw as well as the feedback device (encoder, linear scale etc.). Lead accuracy of the screw, resolution of encoders, and ability of the controller must combine to produce the desired accuracy. In most microscopes, the accuracy is very high and can be calibrated.

Repeatability: Repeatability is the ability of a device to reach a specific location multiple times. It does not take into account the trueness of the position but rather the ability to go back to the same position. Many times the actuator will follow a slightly bowed or twisted path due to imperfect construction. The repeatability is the measure of uncertainty relative to a given actuator movement. The repeatability cannot be calibrated, but it can be measured for any microscope stage. Each time the stage controller requests a position value, the actual resulting position is within repeatability of the requested value.

In a grid tiling, the positions (x, y), that the stage will go to, have an uncertainty equal to the stage repeatability $(x \pm r, y \pm r)$. However, translations (dx, dy) computed in the vertical or horizontal directions between consecutive tiles are differences between respective positions. Thus, the uncertainty on the computed translation values is $2 \times repeatability$ $(dx \pm 2r_x, dy \pm 2r_y)$. The actuators are independent from one another. However, since they are very similar and from the same manufacturer we will assume that $r_x = r_y = r$.

When computing vertical translations between two consecutive tiles, the dx and dy components will correspond to the projection of the stage displacement on the camera coordinate systems (which might have a fixed rotational angle) with an uncertainty equal to $2 \times repeatability$. We use this information to estimate the stage repeatability from the computed translations.

3 Translation Computation

Algorithm 1 details the relative translation computation phase which loops over all image tiles in the Image Grid and, when applicable, computes two translations: one for the image tile and its western neighbor and another for the image tile and its northern neighbor[1]. Each translation consists of a horizontal displacement (x), a vertical displacement (y), and a normalized cross correlation (ncc) forming a translation tuple $\langle ncc, x, y \rangle$.

Translation Computation (Algorithm 1) Method Call Hierarchy:

1. Phase Correlation Image Alignment Method (Algorithm 2)
    (a) Peak Correlation Matrix (Algorithm 3)
    (b) Multi-Peak Max (Algorithm 4)
    (c) Interpret Translations (Algorithm 5)
        i. Extract Overlap Subregion (Algorithm 7)

[1] One can develop easily a modified version of the algorithm that computes translations for a tile's eastern and southern neighbors.

ii. Normalized Cross Correlation (Algorithm 6)

Algorithm 1: Relative-Translation-Computation

```
// Function translationComputation(imgGrid)
Input: Image Grid
Output: two arrays of translation tuples ⟨ncc, x, y⟩
begin
    foreach I ∈ Image Grid do
        T_w[I] ← pciam(I#west, I)                    // when applicable
        T_n[I] ← pciam(I#north, I)                   // when applicable
    end
    return T_w, T_n
end
```

3.1 PCIAM

The function `pciam` (Algorithm 2) implements the Phase Correlation Image Alignment Method which operates on two image tiles, I_1 & I_2. This is a direct image alignment method, a version of Kuglin and Hines' phase correlation image alignment method [3] that is modified to use normalized cross correlation coefficients as described by Lewis [4, 5].

The `pciam` function uses three helper-functions, `pcm`, `multiPeakMax`, and `interpretTranslation`. The function `pcm` (Algorithm 3) computes the Peak Correlation Matrix (PCM) for the input pair of images, I_1 & I_2. PCM is a 2D matrix of double precision values $\in [-1, 1]$. The function `multiPeakMax` (Algorithm 4) finds $n = 2$ peaks within the PCM. The value $n = 2$ was manually selected based on experimental testing performed on the available datasets and was sufficient to achieve our required accuracy. The number of peaks to test in the PCM matrix is an advanced parameter that can be adjusted based on the dataset being stitched.

There is no consensus in literature about the number of peaks to check for the PCIAM algorithm. The Fiji Grid Collection Stitching plugin by Stephan Preibisch checks 5 peaks by default [8]. The paper by Yang Yu et al. checks 8 peaks [11]. Other papers advocate checking only a single peak.

The PCM peaks form a set of potential translations between the two images. Each translation peak is tested to determine the normalized cross correlation coefficient produced by that translation. The function `interpretTranslation` (Algorithm 5) determines the correct interpretation of a given PCM peak. Each peak has several possible interpretations due to the periodicity of the Fourier transform. For each possible interpretation the normalized cross correlation value is computed. The interpretation that maximizes the ncc value is correct.

Algorithm 2: Phase-Correlation-Image-Alignment-Method

```
// Function pciam(I_1, I_2)
Input: two images—same size!
Output: translation tuple ⟨ncc, x, y⟩
begin
    PCM ←pcm(I_1,I_2)                                  // Compute PCM matrix
    n ← 2                                              // number of peaks to find
    Peaks ←multiPeakMax(PCM, n)                        // Perform Multi-Max Peak Search
    // Peaks is a set of tuples, each tuple containing ⟨x, y, val⟩
    foreach peak ∈ Peaks do
        ⟨peak.val, peak.x, peak.y⟩ ← interpretTranslation(I_1, I_2, peak.x, peak.y)
    end
    ⟨ncc, x, y⟩ ← max(Peaks)                           // Find peak with maximum peak.ncc
    return ⟨ncc, x, y⟩
end
```

3.2 Peak Correlation Matrix

The function `pcm` (Algorithm 3) computes the PCM for the two input image tiles. The PCM matrix is the same size as both input images. The value of PCM at index (i, j) (row-column matrix coordinates) can be interpreted as the likelihood that the two input images, when offset by i pixels vertically and j pixels horizontally, form a single scene. Figure 2 shows two individual images that overlap to form a single scene when the second image is translated to the right by roughly 400 pixels. For this example, the PCM would have its maximum value at roughly $(0, 400)$ indicating a translation between the images of 0 pixels vertically and 400 pixels horizontally.

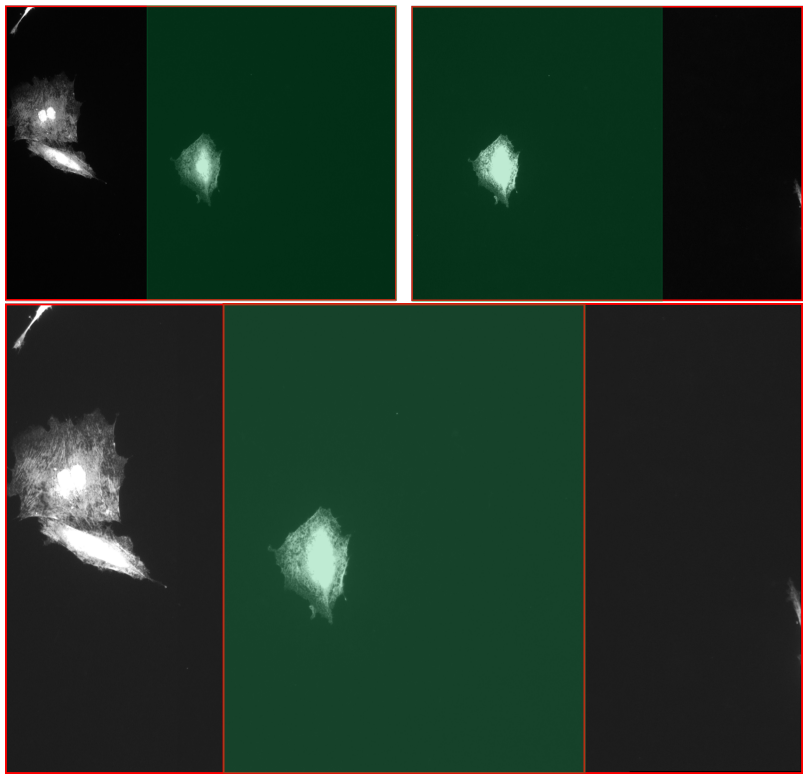

Figure 2: Example pair of images with different illuminations outlined in red whose content forms a single scene. Overlapping region highlighted in green.

Algorithm 3: Peak-Correlation-Matrix

`// Function pcm(`I_1, I_2`)`
Input: two images—same size!
Output: 2D double precision PCM matrix

begin
  $F_1 \leftarrow$ `fft2D(`I_1`)`
  $F_2 \leftarrow$ `fft2D(`I_2`)`
  $FC \leftarrow F_1 . * \overline{F_2}$ `// Operator` $\overline{F_2}$ `is complex conjugate`
  $PCM \leftarrow$`ifft2D(`FC`./abs(`FC`))` `// Operators .* and ./ are element-wise`
  return PCM
end

Under ideal conditions the PCM matrix (2D double precision values $\in [-1, 1]$) contains a single peak (impulse) at the correct translation [10, 12]. Figure 2 shows an example. However, optical microscopy can generate images with few distinguishing features available to guide image registration. Thus, for robustness, we need to account for PCM matrices lacking a single well defined peak. Figure 3 shows a well behaved PCM with a single peak displayed

as an image. Figure 4 shows the same PCM displayed as a surface plot to highlight the structure of the matrix. Figure 5 shows a noisy PCM, lacking a single well defined peak, displayed as an image. Figure 6 shows the same noisy PCM displayed as a surface plot.

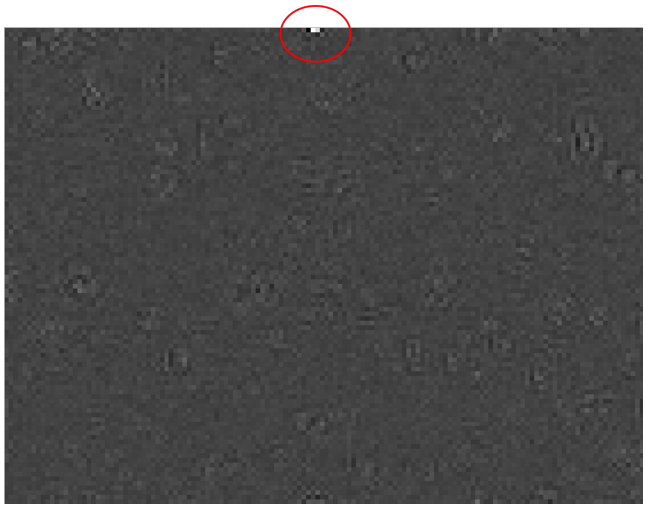

Figure 3: High SNR PCM matrix displayed as an image.

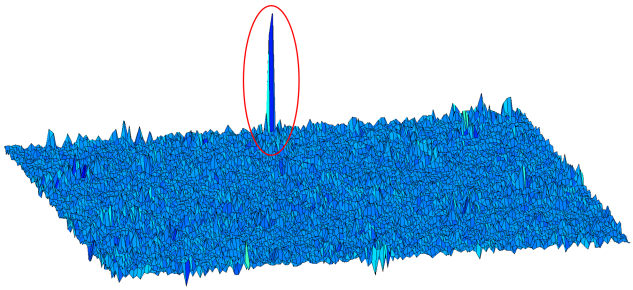

Figure 4: High SNR PCM matrix displayed as a surface plot.

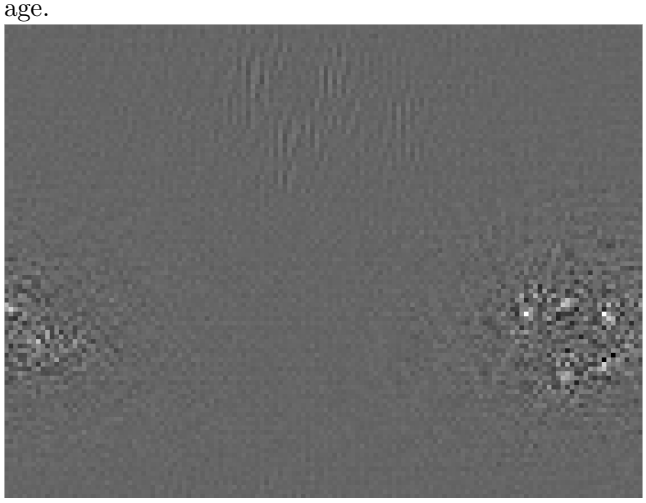

Figure 5: Low SNR (noisy) PCM matrix displayed as an image.

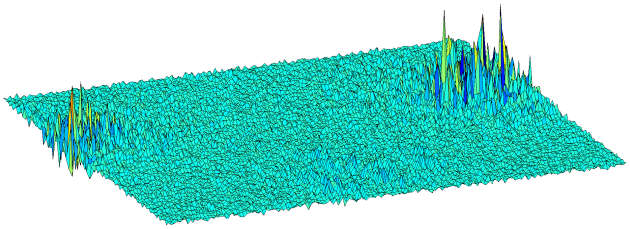

Figure 6: Low SNR (noisy) PCM matrix displayed as a surface plot.

The function `pcm` relies on forward and backward Fourier transforms to compute the PCM. The function `fft2D` is the forward, two dimensional, Fourier transform of a 2D matrix (input image). The function `ifft2D` is the backward (inverse) Fourier transform of the frequency domain matrix FC. Both are assumed to be library functions. The MIST algorithm was developed in MATLAB which provides 2D fft libraries. The ImageJ/Fiji plugin developed in Java relies on FFTW for native compiled 2D FFT libraries [1], cuFFT for CUDA GPU FFT libraries[7], and a Java implementation of a FFT approximation from Dave Hale [2]. Note: Using the CUDA libraries requires the user to download and install the CUDA toolkit [6]. Documenting discrete fast Fourier transform algorithms is beyond the scope of this document, however for the interested reader the authors suggests the FFTW homepage as a starting point [1].

3.3 Multi-Peak Max Search

Under ideal conditions the PCM matrix would contain a single peak defining the correct translation. To account for non-ideal PCM cases, we search for multiple peaks within the PCM and select the peak that maximizes the implied normalized cross correlation value.

The function `multiPeakMax` (Algoritm 4) finds multiple discrete peaks within the PCM. A peak is defined as the (x, y) location of an element in the PCM. A peak, (x, y) matrix location, is equivalent to a translation of x pixels horizontally and y pixels vertically between the two input images used to compute the PCM. The set of peaks `multiPeakMax` finds is defined as the highest $(n = 2)$ PCM values. Potential (x, y) displacements are magnitude values (always positive by definition) for which direction needs to be determined. A common constraint on the PCM matrix is ensuring a peak is a local maxima within a specified radius, generally a $3x3$ neighborhood, or radius of $\sqrt{2}$ from the PCM peak [9][11]. However, in experimenting with synthetically created image tile grids we found that enforcing this local maxima was detrimental to the percentage of correct translations that were found by the PCIAM technique. Without the local maxima constraint, the multi-peak-max algorithm reduces to a single selection of the highest n values (a variation of the k-select algorithm).

Algorithm 4: Multi-Peak-Max

```
// Function multiPeakMax(PCM, n)
Input: 2D double precision matrix, number of peaks to find
Output: set of tuples; each tuple is a peak containing ⟨x, y⟩
begin
    Peaks ← ∅                                              // initialize output set
    foreach pixel ∈ PCM do
        // pixel is a tuple containing ⟨x, y, val⟩
        if pixel.val > min(Peaks.val) then
            if pixel ∉ Peaks then
                // remove element with min(Peaks.val) from Peaks
                // add pixel to Peaks
            end
        end
    end
    return Peaks
end
```

3.4 Translation Interpretation

Once the set of peaks has been found using `multiPeakMax` the correct implied translation needs to be determined for each peak. Fourier transforms are periodic in nature resulting in four possible interpretations of any magnitude displacement tuple $\langle x, y\rangle$ extracted from the PCM matrix. Given the image width W and height H, the four interpretations are $[(x, y), (x, H-y), (W-x, y), (W-x, H-y)]$. The function `interpretTranslation` (Algorithm 5) determines which of the potential interpretations (translations) generates the maximum normalized cross correlation value. Figure 7 shows the four possible translations that result from Fourier transform periodicity. Each translation maps I_2 into I_1's coordinate space given a magnitude displacement tuple $\langle x, y\rangle$.

In the general case, the translation from I_1 to I_2 can be any $\langle x, y\rangle$ so long as the two images overlap. Therefore, given an input $\langle x, y\rangle$, where x and y are positive (by definition), we need to check 16 possible translations for validity to find the interpretation that has the maximum normalized cross correlation. The 16 combinations arise from the four Fourier transform possibilities, $[(x, y), (x, H - y), (W - x, y), (W - x, H - y)]$, and the four direction possibilities, $(\pm x, \pm y) = [(x, y), (x, -y), (-x, y), (-x, -y)]$.

For this algorithm a priori knowledge limits the general case of 16 combinations per image pair to 8 combinations. From the image grid we know that if I_1 and I_2 form a left-right pair I_2 will always be to the right of I_1. This reduces the search space to the four possible Fourier transform combined with $(x, \pm y)$. If I_1 and I_2 form an up-down pair,

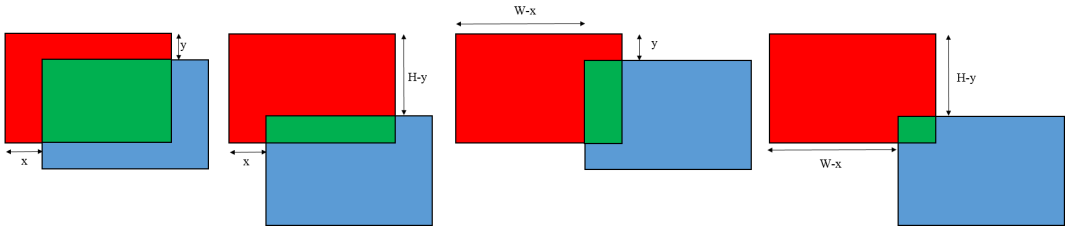

Figure 7: Translation interpretations resulting from Fourier transform periodicity. I_1 in red, I_2 in blue, and the overlap region in green.

I_2 will always be below I_1. This reduces the search space to the four Fourier transform possibilities combined with $(\pm x, y)$.

In order to simplify the pseudo-code, Algorithm 5 presents the general case search of all 16 possible translation interpretations.

Algorithm 5: Interpret-Translation

`// Function interpretTranslation(`I_1, I_2, x_{in}, y_{in}`)`
Input: two images—same size!, magnitude (x, y) displacement values
Output: translation tuple

begin
  $ncc \leftarrow -\infty$ `// Initialize outputs`
  $x \leftarrow 0$, $y \leftarrow 0$
  `// find the translation interpretation with the highest` ncc
  foreach $x_{mag} \in \{x_{in},$ `width(`I_1`)`$-x_{in}\}$ **do**
    foreach $y_{mag} \in \{y_{in},$ `height(`I_1`)`$-y_{in}\}$ **do**
      foreach $x_{sign} \in \{-1,\ 1\}$ **do**
        foreach $y_{sign} \in \{-1,\ 1\}$ **do**
          $subI_1 \leftarrow$`extractOverlapSubregion(`$I_1, (x_{mag} * x_{sign}), (y_{mag} * y_{sign})$`)`
          `// the translation from` I_2 `to` I_1 `is the inverse of the translation from` I_1 `to` I_2
          $subI_2 \leftarrow$`extractOverlapSubregion(`$I_2, -(x_{mag} * x_{sign}), -(y_{mag} * y_{sign})$`)`
          $ncc\_val \leftarrow$`ncc(`$subI_1, subI_2$`)` `// compute normalized cross correlation`
          if $ncc\_val > nnc$ **then**
            $ncc \leftarrow ncc\_val$
            $x \leftarrow x_{sign} * x_{mag}$
            $y \leftarrow y_{sign} * y_{mag}$
          end
        end
      end
    end
  end
  return $\langle ncc, x, y \rangle$
end

The function `ncc` (Algorithm 6) takes as input two arrays of the same size and returns the normalized cross correlation coefficient factor relating their similarity. This function is used to compare equally sized subregions of images.

The function `extractOverlapSubregion` (Algorithm 7) takes as input an image with a translation $\langle x, y \rangle$ and returns the subregion implied if you take a sliding window the size of the image and translate it by x pixels horizontally and y pixels vertically. Figure 8 shows an image with its translated view window, highlighting the extracted subregion in green. This function extracts the subregion of the image that would overlap a hypothetical neighboring image that is offset by the translation $\langle x, y \rangle$.

Algorithm 6: Normalized-Cross-Correlation

```
// Function ncc(I_1, I_2)
```
Input: two images—same size!
Output: cross correlation factor (`double`)

begin
  $I_1 \leftarrow I_1 - \texttt{mean}(I_1)$ `// Center both vectors`
  $I_2 \leftarrow I_2 - \texttt{mean}(I_2)$
  $n \leftarrow I_1 \cdot I_2$ `// dot product`
  $d \leftarrow |I_1| * |I_2|$ `// product of L2-norms`
  return n/d
end

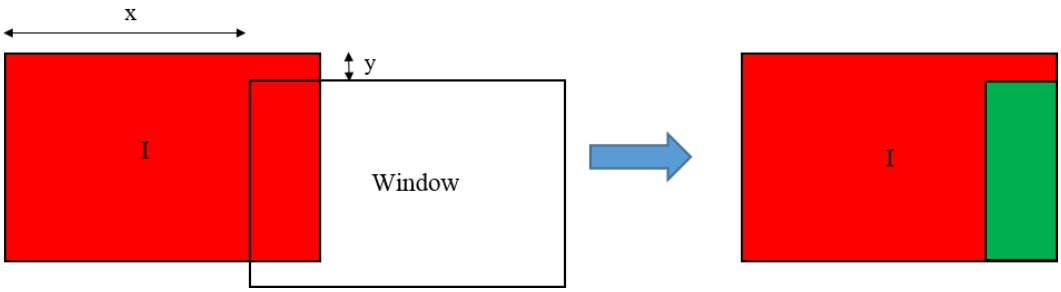

Figure 8: The input image I has its view window translated (x, y) pixels to extract a subregion, shown in green.

Algorithm 7: Extract-Overlapping-Image-Subregion

```
// Function extractOverlapSubregion(I, x, y)
```
Input: image, translation (x, y)
Output: subregion of the image

begin
  `// get the size of the image`
  $H \leftarrow \texttt{height}(I)$
  $W \leftarrow \texttt{width}(I)$
  if $(|x| >= W) \vee (|y| >= H)$ **then**
    return *null* `// if no overlap`
  end
  `// create subregion and constrain to valid image coordinates`
  $x_{start} \leftarrow \texttt{max}(0, \texttt{min}(x, W-1))$
  $x_{end} \leftarrow \texttt{max}(0, \texttt{min}(x+W, W-1))$
  $y_{start} \leftarrow \texttt{max}(0, \texttt{min}(y, H-1))$
  $y_{end} \leftarrow \texttt{max}(0, \texttt{min}(y+H, H-1))$
  `// return the subregion from` I_1
  return $I[y_{start} : y_{end}, x_{start} : x_{end}]$
end

4 Translation Optimization

At the completion of the translation computation phase, each image tile in the image grid has a translation relative to its western neighbor and another translation relative to its northern neighbor, if those neighbors exist.

The goal of the translation optimization phase is to filter and correct image tile translations using our stage actuator movement model. Our model leverages knowledge of how a motorized XY-Stage operates to introduce constraints on the translations. For model details and explanation see Section 2. The translation optimization phase pseudo-code is given in Algorithm 8.

Translation Optimization (Algorithm 8) Method Call Hierarchy:

1. Build Stage Model (Algorithm 9)
   (a) Compute Image Overlap (Algorithm 10)
   (b) Compute Stage Repeatability (Algorithm 13)
2. Apply Stage Model (Algorithm 14)
   (a) Filter by Overlap and Correlation (Algorithm 15)
   (b) Filter by Outliers(Algorithm 16)
   (c) Filter by Repeatability (Algorithm 17)
   (d) Replace Invalid Translations with Estimates
      i. Replace Invalid Translations (Algorithm 18)
      ii. Estimate Empty Row/Columns (Algorithm 19)
3. Bounded Translation Refinement (*ncc* hill climbing) (Algorithm 21)

As part of the translation optimization, the algorithm will estimate the stage model parameters: (1) the estimated uncertainty in the percent overlap between images; (2) the repeatability of the mechanical stage actuators (NORTH and WEST); (3) and the percent overlap between images (NORTH and WEST). The percent overlap uncertainty (*pou*) between images defaults to 3% but can be overridden by the user. Both the image overlap (per direction), and the stage actuator repeatability (r) are estimated from the translations be default. Any of these estimated values can be overridden by providing the correct value as a parameter.

The translation optimization phase filters and corrects the north and west translations independently; building the stage model for each direction. The translation optimization then filters out the translations which do not fit into the stage model, replacing them with estimates derived from the stage model. The translations are then refined by using a bounded hill climbing search to maximize the normalized cross correlation values. The hill climbing search allows translations to migrate to a local maximum *ncc* value in the normalized cross correlation (NCC) surface. The NCC surface is a matrix where the value at element $NCC[i, j]$ is the result of computing the normalized cross correlation coefficient between the overlapping subregions of an image pair given the translation ($x = j, y = i$). The psuedo-code for computing a single element of NCC surface is given here.

$NCC[i, j] \leftarrow$`ncc(extractOverlapSubregion(`I_1, j, i`),extractOverlapSubregion(`$I_2, -j, -i$`))`;[2]

Iterating over all valid i & j would fill in the surface producing an example like Figure 9. Using hill climbing with normalized cross correlation as the cost function allows our stitching algorithm to find pixel level optimal translations, presuming a reasonable starting translation estimate.

The hill climbing pseudo-code is given in Algorithm 21.

[2]The translation for I_2 is the inverse of the translation for I_1 because the translation from I_2 to I_1 is the opposite of the translation from I_1 to I_2. Translation tuples use (x, y) coordinates and matrices use (i, j) coordinates; the conversion is $x = j$, $y = i$

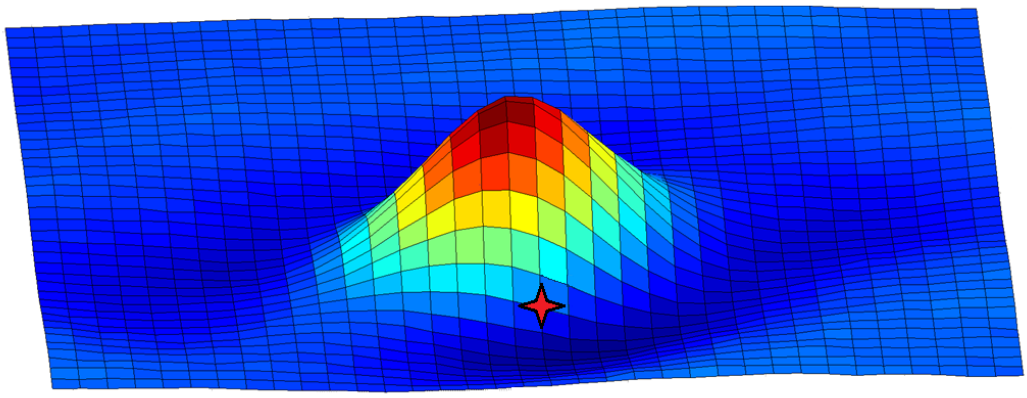

Figure 9: Example NCC matrix (surface) upon which hill climbing operates, with the hill climb starting point (estimated translation) marked with a star.

Algorithm 8: Translation-Optimization

`// Function translationOptimization`$(imgGrid, T_w, T_n)$
Input: Image grid, two 2D arrays of translation tuples
Output: Two 2D arrays of corrected translation tuples

begin

$pou \leftarrow 3$
$model_w \leftarrow$`buildStageModel`$(imgGrid, T_w, pou, WEST)$ `// build west stage model`
$model_n \leftarrow$`buildStageModel`$(imgGrid, T_n, pou, NORTH)$ `// build north stage model`

$T_w \leftarrow$`applyStageModel`$(imgGrid, T_w, model_w, pou, WEST)$ `// filter west translations`
$T_n \leftarrow$`applyStageModel`$(imgGrid, T_n, model_n, pou, NORTH)$ `// filter north translations`

$repeatability \leftarrow 2*$`max`$(model_w.r, model_n.r) + 1$
$model_w.r \leftarrow repeatability$
$model_n.r \leftarrow repeatability$

$T_w \leftarrow$`refineTranslations`$(imgGrid, T_w, model_w, WEST)$ `// refine west translations`
$T_n \leftarrow$`refineTranslations`$(imgGrid, T_n, model_n, NORTH)$ `// refine north translations`

return T_w, T_n

end

The function `buildStageModel` (Algorithm 9) uses the pairwise translations between images to estimate the overlap and stage repeatability per direction (NORTH and WEST). The function `applyStageModel` (Algorithm 14) uses the stage model and filters the pairwise translations to remove those with low confidence. The invalid translations are then replaced with an appropriate estimate based on the stage model. The function `refineTranslations` (Algorithm 20) takes the results of translation filtering and refines those translations to a local maximum using normalized cross correlation as the cost function. This translation refinement is bounded by the stage model to ensure that the resulting translations are reasonable.

4.1 Build Stage Model

The goal of translation optimization is to filter and correct image tile translations using our stage actuator movement model. Our model leverages knowledge of how a motorized XY-Stage operates to introduce constraints on the translations.

The algorithm estimates the following stage model parameters: (1) the estimated uncertainty in the overlap between images as a percentage; (2) the repeatability of the mechanical stage actuators (NORTH and WEST); and (3) the

overlap between images as a percentage (NORTH and WEST). The percent overlap uncertainty (pou) between images defaults to 3% but can be overridden by the user. Both the image overlap (per direction), and the stage actuator repeatability (r) are estimated from the translations by default. Any of these estimated values can be overridden by providing the correct value as a parameter.

It is useful to think of the translations as being a matrix the same size and shape as the image tile grid. This results in a matrix of vertical translations (each translation containing $\langle x, y, ncc \rangle$) and a matrix of horizontal translations.

The function `buildStageModel` (Algorithm 9) uses the pairwise translations between images to estimate the overlap and stage repeatability per direction (NORTH and WEST).

Algorithm 9: Build Stage Model

`// Function buildStageModel(`$imgGrid, T, pou, direction$`)`
Input: Image grid, 1D array of translations,percent overlap uncertainty,direction
Output: stage model object containing repeatability and overlap

begin

`// compute image overlap`
$overlap \leftarrow$`computeOverlap(`$imgGrid, T, direction$`)`

`// limit to valid values`
$overlap \leftarrow$`max(`pou, `min(`$overlap, 100 - pou$`))`

`// compute stage actuator repeatability`
$repeatability \leftarrow$`computeRepeatability(`$imgGrid, T, overlap, pou, direction$`)`

$stageModel \leftarrow [repeatability, overlap]$

return $stageModel$

end

4.1.1 Compute Image Overlap

The function `computeImageOverlap` (Algorithm 10) determines an estimated percent overlap between images given a set of translation tuples. For example, given a set of NORTH translations this function computes the estimated overlap between image tiles and their northern neighbors. In other words, the stage model image overlap is computed by fitting a model to the primary travel direction translations (x for WEST and y for NORTH) using maximum likelihood estimation. When range filtering translations a percent overlap uncertainty (pou) margin is included to account for variations in image overlap. This uncertainty margin can be provided by the user or left as default.

Algorithm 10: Compute Image Overlap

```
// Function computeImageOverlap(imgGrid, T, direction)
```
Input: Image grid, 1D array of translations,direction
Output: percent overlap

begin
  // Maximum Likelihood Estimation is used to fit a model to the translations
  // finding MLE uses variant of stochastic local search (hill climbing)
  if $direction = NORTH$ **then**
    $range \leftarrow$// image tile height
    $T \leftarrow 100 \times T/range$ // scale translations into $[0, 100]$
  else
    $range \leftarrow$// image tile width
    $T \leftarrow 100 \times T/range$ // scale translations into $[0, 100]$
  end
  // model contains $[probUniform, mu, sigma, likelihood]$
  $bestModel \leftarrow [0, 0, 0, -\inf]$
  // termination condition: optimization stalls for $maxStallCount$ iterations
  $maxStallCount \leftarrow 20$
  $stallCount \leftarrow 0$
  while $stallCount < maxStallCount$ **do**
    // generate random MLE model search starting point
    // rand() generates a uniform random value $\in [0, 1]$
    $model \leftarrow [100*$rand(), $100*$rand(), $100*$rand(), $NaN]$
    // perform percent resolution hill climbing
    $model \leftarrow$percentileResolutionHillClimb($model, T$)
    if $model.likelihood > bestModel.likelihood$ **then**
      $bestModel \leftarrow model$
      $stallCount \leftarrow 0$
    else
      $stallCount \leftarrow stallCount + 1$
    end
  end
  $overlap \leftarrow 100 - bestModel.mu$ // mu is the average translation in percent
  return $overlap$
end

The function `percentileResolutionMleHillClimb` (Algorithm 11) performs hill climbing in the stage model parameter space in order to find the MLE stage model with the highest likelihood.

Algorithm 11: Perform Percentile Resolution MLE Hill Climbing

```
// Function percentileResolutionMleHillClimb(model, T)
Input: stage model, 1D array of translations
Output: stage model
begin
    // get the valid translation range for the current direction
    done ← false
    while ¬done do
        // neighbors differ from model by distance of 1 in a single dimension
        foreach neighbor do
            // check all neightbors and select the one with the highest likelihood
            temp ← model
            if neighbor is within bounds then
                likelihood ←computeMleLikelihood(model, T)
                if likelihood > temp.likelihood then
                    temp ← neighbor
                end
            end
        end
        if temp = model then
            done ← true
        else
            model ← temp
        end
    end
    return model
end
```

The function `computeMle` (Algorithm 12) computes the likelihood of a stage model given a set of translations. The function requires the stage model in question and a vector of translations. The stage model contains the probability that a translation comes from the uniform distribution, the mean of the expected normal distribution of valid translations, and the sigma of the expected normal distribution of valid translations. This model assumes that a vector of translations from the PCIAM can be split into two subsets. The subset of valid translations is expected to be approximately normally distributed. The subset of invalid translations is expected to be approximately uniformly distributed.

Algorithm 12: Compute MLE

`// Function computeMle(`$model, T$`)`
Input: stage model, 1D array of translations
Output: model likelihood

begin
  `// by definition` $\forall T \in [0, 100]$
  `// by definition the model parameters (`$probUniform, \mu, \sigma$`)` $\in [0, 100]$

  `// initialize likelihood`
  $likelihood \leftarrow 0$
  foreach $t \in T$ **do**
    `// likelihood that` t `belongs to` $N(model.\mu, model.\sigma)$
    $normLikelihood \leftarrow \frac{1}{model.\sigma\sqrt{2\pi}} e^{-(t-model.\mu)^2/2model.\sigma^2}$
    `// likelihood that` t `belongs to uniform distribution`
    $uniformLikelihood \leftarrow \frac{1}{100}$
    `// Convert` $model.probUniform$ `from` $[0, 100]$ `to` $[0, 1]$
    $p \leftarrow model.probUniform/100$
    `// Compute likelihood update`
    $l \leftarrow p \times uniformLikelihood + (1 - p) \times normLikelihood$
    $likelihood \leftarrow likelihood$+`log(abs(`$l$`))`
  end
  return $likelihood$
end

4.1.2 Compute Stage Repeatability

The function `computeRepeatability` (Algorithm 13) determines the XY-Stage actuator mechanical repeatability. For a given translation direction (NORTH or WEST), the x & y repeatability are computed independently. The repeatability for that translation direction (NORTH or WEST) is the max of the independent x & y repeatability values. The max of the two repeatability values is selected to create a single repeatability value per direction the encompass both repeatability values. The NORTH and WEST repeatability values are eventually combined into a single repeatability value to bound the NCC Hill Climbing Translation Optimization 4. See Section 2 for more detail on the stage repeatability.

Algorithm 13: Compute Stage Repeatability

```
// Function computeRepeatability(imgGrid, T, overlap, pou, direction)
Input: Image grid, 2D array of translation tuples,overlap,percent overlap uncertainty,direction
Output: stage repeatability
begin
    // filter translations by overlap ± pou and correlation > 0.5
    validTranslations ←filterByOverlapAndCorrelation(imgGrid, T, overlap, pou, direction)
    // filter translation outliers
    validTranslations ←filterOutliers(validTranslations, direction)
    if direction = NORTH then
        r_x ← ⌈(max(validTranslations.x)−min(validTranslations.x))/2⌉        // x repeatability
        r_y ← 0                                                              // y repeatability
        for j ← 0 to width(T) − 1 do
            min_val ← ∞, max_val ← −∞
            for i ← 0 to height(T) − 1 do
                if T[i, j] ∈ validTranslations then
                    min_val ←min(min_val, T[i, j].y)
                    max_val ←max(max_val, T[i, j].y)
                end
            end
            column_range ← (max_val − min_val)
            r_y ← max(r_y, (column_range/2))
        end
    else
        r_y ← ⌈(max(validTranslations.y)−min(validTranslations.y))/2⌉        // y repeatability
        r_x ← 0                                                              // x repeatability
        for i ← 0 to height(T) − 1 do
            min_val ← ∞, max_val ← −∞
            for j ← 0 to width(T) − 1 do
                if translations[i, j] ∈ validTranslations then
                    min_val ← min(min_val,T[i,j].x)
                    max_val ← max(max_val,T[i,j].x)
                end
            end
            row_range ← (max_val − min_val)
            r_x ← max(r_x, (row_range/2))
        end
    end
    return max(r_x, r_y)
end
```

4.2 Apply Stage Model

The function `applyStageModel` (Algorithm 14) filters and corrects a 2D grid of translation tuples using the stage model constructed in `buildStageModel` (Algorithm 9). Each translation direction (NORTH, WEST) is corrected independently. Translations are filtered orthogonal to the primary direction of travel, the x components of NORTH translations and the y components of WEST translations.

To extract an upper limit for the stage actuator repeatability from the computed translations this algorithm relies on the fact that the image tile grid forms a regular grid. The authors define a regular grid such that the distance between adjacent image tiles is approximately equal within a given direction. For NORTH translations the y components are approximately equal, the x components are approximately equal, and the x components are close to zero. For WEST translations the x components are approximately equal, the y components are approximately equal, and the y components are close to zero.

Consider a single column of vertical translations from the image tile grid. To produce that column of images the microscope mechanical stage is moved to a series of locations, each of which has the same x value and a different y value. Similarly for a row of horizontal translations the stage is moved to a series of locations, each of which has the same y value and a different x value. Thus for a matrix of NORTH translations one can look at the horizontal (x) components to estimate an upper bound for the mechanical stage repeatability. For a matrix of WEST translations one can look at the vertical (y) components to estimate an upper bound for the stage repeatability.

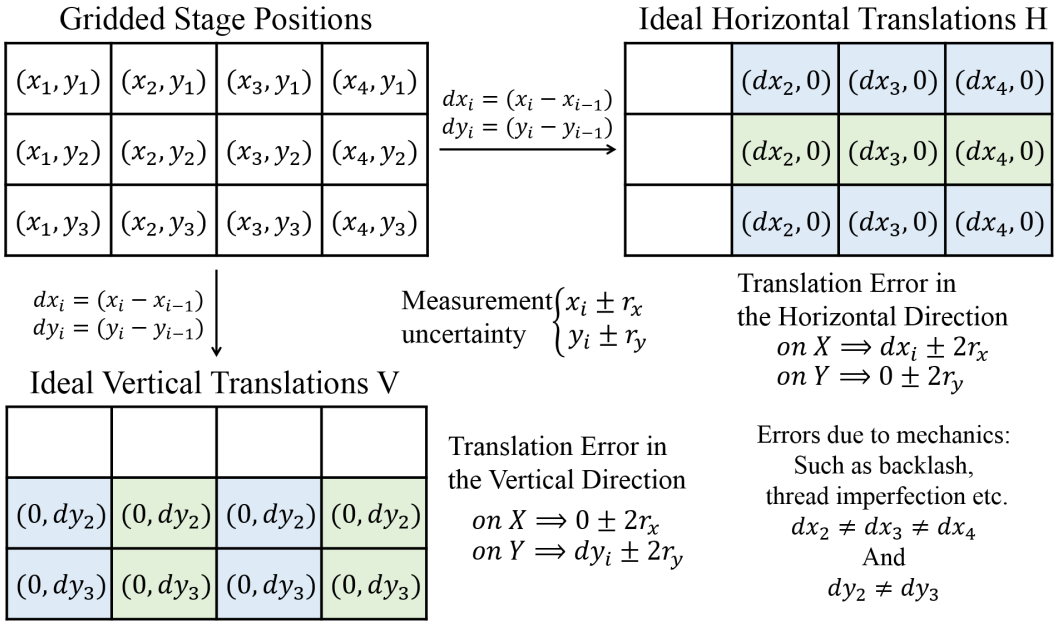

Figure 10: Grid of stage displacements as observed by the camera.

The goal of `applyStageModel` is to filter the translations into two sets, valid translations and invalid translations, replacing the invalid translations with estimates derived from the valid set. The filtered and corrected translations are later used in the hill climbing translation refinement stage.

The stage model repeatability is used to re-filter the translations and reclaim any that should be considered valid. Translations are reclaimed if they fall within $\pm repeatability$ of the median valid translation for that row or column and have a valid *ncc* value ($ncc >= 0.5$). This reclaiming step provides a stringent initial range filter, increases the accuracy and robustness of the repeatability computation, and retains the correct pool of valid translations. The re-filter and reclaim function, `filterByRepeatability`, is documented in Algorithm 17.

The function `replaceInvalidTranslations` (Algorithm 18) alters the translation estimate of every invalid translation which has at least one valid translation in its row (for NORTH) or column (for WEST). An invalid translation is replaced with the median of the valid translations for that row or column. This estimated translation becomes the starting point for the hill climbing search performed later and prevents the invalid translations from corrupting the hill climbing search.

The function `estimateEmptyRowColumn` (Algorithm 19) creates an estimated translation for any row (for NORTH) or column (for WEST) without any valid translations. It replaces all translations in the row or column with the estimate to provide the hill climbing with a reasonable starting point.

Algorithm 14: Apply Stage Model

```
// Function applyStageModel(imgGrid, T, model, pou, direction)
Input: Image grid, 2D array of translation tuples, stage model, direction, percent overlap uncertainty
Output: set of valid translations
begin
    // Filter the translations by overlap and correlation
    validTranslations ←filterByOverlapAndCorrelation(imgGrid, T, model.overlap, pou, direction)
    // Filter by translation outliers
    validTranslations ←fitlerOutliers(T, direction)
    // Filter and reclaim the translations by stage repeatability
    validTranslations ←filterByRepeatability(imgGrid, T, validTranslations, model.r, direction)
    if validTranslations = ∅ then
        // compute estimated translation from overlap
        // set all translation to the estimate
    else
        // replace invalid translations with the median valid translation per row/column
        T ←replaceInvalidTranslations(T, validTranslations, direction)
        // estimate translation for any completly empty row/column
        T ←estimateEmptyRowColumn(imgGrid, T, validTranslations, direction)
    end
    return T
end
```

4.2.1 Filter By Overlap and Correlation

The function `filterByOverlapAndCorrelation` (Algorithm 15) range filters the translations. Range filtering the translations is done in the primary movement direction. Horizontal translations have their x component filtered. Vertical translations have their y component filtered. This range filtering is done in the primary movement direction because, assuming a regular image grid, the primary movement component of all translations should be approximately equal. In an ideal case the NORTH translation y components will all be equal and the x components will all be zero. In an ideal case the WEST translation y components will all be zero and the x components will all be equal. By computing an estimated image overlap and then range filtering the translations any translations that are not in the correct approximate range are discarded.

Valid translations fall within a percent overlap uncertainty (*pou*) of the estimated image overlap. In other words, translations that result in an image overlap within a percent overlap uncertainty (*pou*) of the computed image overlap (from Algorithm 10) are considered valid. For example, if a test case had an estimated image overlap of 7% for the north translations and a $pou = 5\%$ then the valid north translation are those that result in an image overlap of $7 \pm 5\%$ and the translations that do not fall in this range are considered invalid.

Translations with low correlation values ($ncc < 0.5$) are removed on the assumption that they are unreliable.

Algorithm 15: Filter Translations by Overlap and Correlation

`// Function filterByOverlapAndCorrelation(`$imgGrid, T, overlap, pou, direction$`)`
Input: Image grid, 2D array of translation tuples,overlap,percent overlap uncertainty,direction
Output: set of valid translations

begin
  `// get image size`
  $H \leftarrow$`height(`$Image$`)`
  $W \leftarrow$`width(`$Image$`)`
  `// determine valid translation range`
  if $direction = NORTH$ **then**
    $range \leftarrow \{(H - (overlap + pou) * H/100), (H - (overlap - pou) * H/100)\}$
  else
    $range \leftarrow \{(W - (overlap + pou) * W/100), (W - (overlap - pou) * W/100)\}$
  end
  `// range filter the translations`
  $validTranslations \leftarrow \emptyset$
  foreach $t \in T$ **do**
    if $direction = NORTH$ **then**
      if $range.min <= t.y <= range.max$ **then**
        $validTranslations \leftarrow validTranslations \cup t$
      end
    else
      if $range.min <= t.x <= range.max$ **then**
        $validTranslations \leftarrow validTranslations \cup t$
      end
    end
  end
  `//` ncc `filter the translations`
  foreach $t \in validTranslations$ **do**
    if $t.ncc < 0.5$ **then**
      $validTranslations \leftarrow validTranslations \setminus t$ `// Operator '\' is set difference`
    end
  end
  return $validTranslations$
end

4.2.2 Filter By Outliers

The function `filterOutliers` (Algorithm 16) removes any valid translations that qualify as outliers. Outliers are defined as translations that are more than $1.5 * inter\_quartile\_distance$ from the median of the valid translations. This definition is taken from the NIST Engineering Statistics Handbook. The $inter\_quartile\_distance$ is defined as the difference between the $3rd$ and $1st$ quartiles of the translations.

Algorithm 16: Filter Translation Outliers

`// Function filterOutliers(`$T, direction$`)`
Input: set of translations tuples,direction
Output: set of valid translations

begin
  $validTranslations \leftarrow \emptyset$
  $w \leftarrow 1.5$ `// w = 1.5 is default statistical outlier threshold (source)`
  if $direction = NORTH$ **then**
    $q2 \leftarrow$`median(`$T.y$`)` `// get median translation (second quartile)`
    $q1 \leftarrow$`median(`$T.y < q2$`)` `// get first quartile`
    $q3 \leftarrow$`median(`$T.y > q2$`)` `// get thrid quartile`
    $iqd \leftarrow$`abs(`$q3 - q1$`)` `// get inter-quartile distance`
    foreach $t \in T$ **do**
      if $(q1 - w \times iqd) < t.y < (q3 + w \times iqd)$ **then**
        `// add the translation to the valid set`
        $validTranslations \leftarrow validTranslations \cup translation$
      end
    end
  else
    $q2 \leftarrow$`median(`$T.x$`)` `// get median translation (second quartile)`
    $q1 \leftarrow$`median(`$T.x < q2$`)` `// get first quartile`
    $q3 \leftarrow$`median(`$T.x > q2$`)` `// get thrid quartile`
    $iqd \leftarrow$`abs(`$q3 - q1$`)` `// get inter-quartile distance`
    foreach $t \in T$ **do**
      if $(q1 - w \times iqd) < t.x < (q3 + w \times iqd)$ **then**
        `// add the translation to the valid set`
        $validTranslations \leftarrow validTranslations \cup translation$
      end
    end
  end
  return $validTranslations$
end

4.2.3 Filter By Repeatability

The function `filterByRepeatability` (Algorithm 17) finds currently invalid translations that have reasonable values and adds them back into the valid translation set. While these translations might not have been required or desirable when building the stage model, if they are within the stage model's repeatability range and have a valid *ncc* value, they should be considered valid translations.

In the previous range filtering steps the computed (or supplied) image overlap and the parameter percent overlap uncertainty defined the valid range. For this filtering the median of the valid translations defines the center point of the valid range. Any translation with a valid *ncc* that exists within a distance *repeatability* of the median of the valid translations is reclaimed. This new range filter: `median`$(validTranslations) \pm repeatability$ is used only to add translations to the valid set that were previously excluded.

Algorithm 17: Filter By Repeatability

`// Function filterByRepeatability(`$imgGrid, translaions, validTranslations, r, direction$`)`
Input: Image Grid, 2D array of translation tuples, set of valid translations, repeatability, direction
Output: set of $validTranslations$ that passed the filter

```
begin
    if direction = NORTH then
        foreach row ∈ imgGrid.rows do
            foreach t ∈ row.translations do
                med_x ← median(row.validTranslations.x)
                med_y ← median(row.validTranslations.y)
                if (t.ncc >= 0.5) ∧ (t.x ∈ [med_x ± r]) ∧ (t.y ∈ [med_y ± r]) then
                    validTranslations ← validTranslations ∪ t
                end
            end
        end
    else
        // direction = WEST
        foreach col ∈ imgGrid.columns do
            foreach t ∈ col.translations do
                med_x ← median(col.validTranslations.x)
                med_y ← median(col.validTranslations.y)
                if (t.ncc >= 0.5) ∧ (t.x ∈ [med_x ± r]) ∧ (t.y ∈ [med_y ± r]) then
                    validTranslations ← validTranslations ∪ t
                end
            end
        end
    end
    return validTranslations
end
```

4.2.4 Replace Invalid Translations

The translation refinement phase performs a hill climbing optimization on the NCC surface in order to maximize the *ncc* value. This hill climbing search requires that each pair of images has a translation that is reasonably close to the correct value so that the hill climbing will converge to the pixel-wise correct answer. For translations in the valid set (from previous filtering steps), nothing needs to be done in order to satisfy this requirement. However, for translations that are not in the valid set, some form of correction or estimation is needed in order to provide the hill climbing search a reasonable starting point.

The function `replaceInvalidTranslations` replaces all invalid translations with estimates derived from the set of valid translations. The translation refinement phase uses the best estimate for an image translation as the starting point for the hill climbing translation optimization. By replacing invalid translations with estimates, the starting point of the hill climbing search is adjusted to increase the likelihood that it will find the correct local maxima in the *ncc* surface. An invalid translation is replaced with the median valid translation for that row/column. If the direction is NORTH, the invalid translation is replaced with the median of the valid translations for that row. If the direction is WEST, the invalid translation is replaced with the median of the valid translations for that column.

Algorithm 18: Replace Invalid Translations

```
// Function replaceInvalidTranslations(translations, validTranslations, direction)
Input: 2D array of translation tuples, valid translations set, direction
Output: updated translations
begin
    if direction = NORTH then
        foreach row ∈ imgGrid.rows do
            if size(row.valid_translations) > 0 then
                foreach t ∈ row.translations do
                    if t ∉ row.validTranslationi then
                        t.x ←median(row.validTranslations.x)
                        t.y ←median(row.validTranslations.y)
                        t.ncc ← −1
                    end
                end
            end
        end
    else
        foreach col ∈ imgGrid.cols do
            if size(col.valid_translations) > 0 then
                foreach t ∈ col.translations do
                    if t ∉ col.validTranslations then
                        t.x ←median(col.validTranslations.x)
                        t.y ←median(col.validTranslations.y)
                        t.ncc ← −1
                    end
                end
            end
        end
    end
    return translations
end
```

4.2.5 Estimate Empty Rows/Columns

The function `estimateEmptyRowColumn` (Algorithm 19) creates an estimated translation for any row or column that has no valid translations. If the direction is NORTH and there exists a row without any valid translations, then a translation must be estimated to replace all of the invalid translations in that row. If the direction is WEST and there exists a column without any valid translation then a translation must be estimated to replace all of the translations in that column. The median of the set of valid translations is used as the estimated replacement translation.

Algorithm 19: Estimate Empty Row Column

`// Function estimateEmptyRowColumn`(*imgGrid*, *translations*, *validTranslations*, *direction*)

Input: Image Grid, 2D array of translation tuples, valid translations set, repeatability, percent overlap uncertainty, overlap, direction

Output: updated set of translations

```
begin
    if direction = NORTH then
        foreach row ∈ imgGrid.rows do
            foreach t ∈ row.translations do
                if t ∉ valid_translations then
                    t.x ←median(validTranslations.x)
                    t.y ←median(validTranslations.y)
                    t.ncc ← −1
                end
            end
        end
    else
        foreach col ∈ imgGrid.cols do
            foreach t ∈ col.translations do
                if t ∉ valid_translations then
                    t.x ←median(validTranslations.x)
                    t.y ←median(validTranslations.y)
                    t.ncc ← −1
                end
            end
        end
    end
    return translations
end
```

4.3 Constrained Translation Refinement

Algorithm 20: Refine Translations

`// Function refineTranslations(`$imgGrid, T, model, direction$`)`
Input: Image grid, 2D array of translation tuples, stage model, direction
Output: 2D array of corrected translation tuples $\langle ncc, x, y \rangle$

begin
  foreach $I \in imgGrid$ **do**
    if $direction = WEST \wedge I$`#west` *exist* **then**
      $t \leftarrow T[I]$
      `// Search range:` $bounds = [y_{min}, y_{max}, x_{min}, x_{max}]$
      $bounds \leftarrow [t.y - r, t.y + r, t.x - r, t.x + r]$
      $\langle ncc, x, y \rangle \leftarrow$`nccHillClimb(`$I, I$`#west`$,bounds, t.x, t.y$`)`
      $T[I] \leftarrow \langle ncc, x, y \rangle$
    end
    if $direction = NORTH \wedge I$`#north` *exist* **then**
      $t \leftarrow T[I]$
      `// Search range:` $bounds = [y_{min}, y_{max}, x_{min}, x_{max}]$
      $bounds \leftarrow [t.y - r, t.y + r, t.x - r, t.x + r]$
      $\langle ncc, x, y \rangle \leftarrow$`nccHillClimb(`$I, I$`#north`$,bounds, t.x, t.y$`)`
      $T[I] \leftarrow \langle ncc, x, y \rangle$
    end
  end
  return T
end

At this point in the algorithm each adjacent pair of images has a translation obtained either from PCIAM or estimation. These translations need to be refined to obtain pixel level image registration accuracy. This is accomplished by performing a hill climbing search with normalized cross correlation as the cost function. The *ncc* cost function takes as input the two images and the translation between them returning a single scalar value representing how well correlated the overlapping regions are. This allows translations to be refined by comparing the *ncc* value of the current translation with the *ncc* values for its 4 neighbors. If the translation in question was of high quality, coming from the PCIAM function, it is likely the current translation is a local maximum in the *ncc* surface and the hill climbing does nothing. On the other hand, estimated translations will most likely be adjusted before finding the local maximum. See Figure 9 for an example *ncc* surface with the hill climbing starting point marked. The pseudo-code for the *ncc* hill climbing is detailed in Algorithm 21.

Algorithm 21: Bounded NCC Hill Climb Translation Refinement

```
// Function nccHillClimb(I_1, I_2, bounds, x, y)
Input: two 2D image arrays—same size, search bounds, (x,y) translation
Output: translation tuple containing ⟨ncc, x, y⟩
begin
    // extract search bounds
    y_min ← bounds[1], y_max ← bounds[2]
    x_min ← bounds[3], x_max ← bounds[4]
    ncc ← −∞
    // while a local max has not been found
    while true do
        bestD_x ← 0
        bestD_y ← 0
        delta_x ← [−1, 1, 0, 0]
        delta_y ← [0, 0, −1, 1]
        // find max in 3x3 four connected neighborhood
        for i ∈ [1, 4] do
            d_x ← delta_x[i]
            d_y ← delta_y[i]
            // if translation is within bounds
            if (x_min <= (x + d_x) <= x_max) ∧ (y_min <= (y + d_y) <= y_max) then
                subI_1 ← extractOverlapSubregion(I_1, (x + d_x), (y + d_y))
                subI_2 ← extractOverlapSubregion(I_2, −(x + d_x), −(y + d_y))
                val ← ncc(subI_1, subI_2)
                if val > ncc then
                    ncc ← val
                    bestD_x ← d_x
                    bestD_y ← d_y
                end
            end
        end
        x ← x + bestD_x
        y ← y + bestD_y
        // if current (x, y) is local max, stop
        if (d_x = 0) ∧ (d_y = 0) then
            return ⟨ncc, x, y⟩
        end
    end
end
```

5 Image Composition

To perform mosaic image composition, each adjacent pair of images must have a relative translation (displacement). These translations form an over-constrained system that one can represent as a directed acyclic graph where vertices are images and edges relate adjacent images. The over-constraint in the system is due to the equivalence between absolute displacements of images and path summations in the graph; these summations which must be path invariant to yield a well-formed image. This phase resolves the over-constraint in the system and computes absolute displacements. It selects a subset of the relative displacements or uses a global optimization approach to adjust them to a path invariant state in the graph. These absolute displacements are used to compose the stitched mosaic image.

A maximum spanning tree is used to resolve the system over-constraint by using the translation normalized cross correlation values as the graph edge weights. The edge weights from valid translations are promoted with a constant value to ensure they are always used before a non-valid (estimated) translation when determining the spanning tree. A maximum spanning tree is used to maximize the set of edge weights used in assembling the tree because the higher the edge weight the higher the confidence in that translation.

With the absolute displacements computed each image tile in the Image Grid has a global (x, y) location in the stitched mosaic image. The stitched image is built by copying the individual image tiles into position within the stitched image and blending the image tiles together.

6 Acknowledgments

This document and the solution approaches outlined in it have benefited from the feedback of many members of the group with Tim Blattner, Joe Chalfoun, and Walid Keyrouz being generous with feedback.

References

[1] Matteo Frigo and Steven˜G. Johnson. The Design and Implementation of {FFTW3}. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[2] Dave Hale. Mines Java toolkit: Java packages for scientific computing, 2014.

[3] C. Kuglin and D. Hines. The Phase Correlation Image Alignment Method. *Proceedings of the 1975 IEEE International Conference on Cybernetics and Society*, pages 163–165, 1975.

[4] J. P. Lewis. Fast Normalized Cross-Correlation. Technical Report 1, Industrial Light & Magic, 1995.

[5] J P Lewis. Fast Template Matching. *Vision Interface*, pages 120–123, 1995.

[6] NVIDIA. CUDA Toolkit, 2014.

[7] NVIDIA. cuFFT Library, 2014.

[8] Stephan Preibisch, Stephan Saalfeld, and Pavel Tomancak. Fast Stitching of Large 3D Biological Datasets. Technical report, Max Planck Institute of Molecular Cell Biology and Genetics, Dresden, Germany, 2007.

[9] Stephan Preibisch, Stephan Saalfeld, and Pavel Tomancak. Globally optimal stitching of tiled 3D microscopic image acquisitions. *Bioinformatics (Oxford, England)*, 25(11):1463–5, June 2009.

[10] Richard Szeliski. Image Alignment and Stitching: A Tutorial. *Foundations and Trends® in Computer Graphics and Vision*, 2(1):1–104, 2006.

[11] Yang Yu and Hanchuan Peng. Automated high speed stitching of large 3D microscopic images. *2011 IEEE International Symposium on Biomedical Imaging: From Nano to Macro*, pages 238–241, March 2011.

[12] Barbara Zitová and Jan Flusser. Image registration methods: a survey. *Image and Vision Computing*, 21(11):977–1000, October 2003.