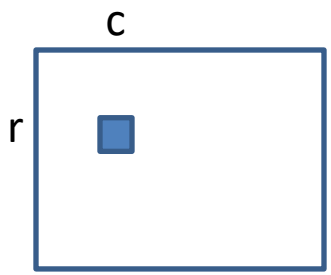
Assignment 2

Creating Panoramas

Step 1: (10 pts) Implement the Harris corner detector.

- Part a. void GaussianBlurImage(**double** *image, int w, int h, double sigma)
- Same as your function for Assignment 1, but the input is a floating point array of size w*h.
- You can use the full 2D kernel.
- You can index a pixel (c,r) by image[r*w + c] to go with the rest of the code for this assignment.



Step 1: (10 pts) Implement the Harris corner detector.

- Part b. void HarrisCornerDetector(QImage image, double sigma, double thres, CIntPt **cornerPts, int &numCornerPts, QImage &imageDisplay)
  - image is the input image
  - sigma is the standard deviation for the Gaussian
  - thres is the threshold for detection corners
  - cornerPts is an array that will contain the returned corner points
  - numCornerPts is the number of points returned
  - imageDisplay – image returned to display for debugging

Step 1: (10 pts) Implement the Harris corner detector.

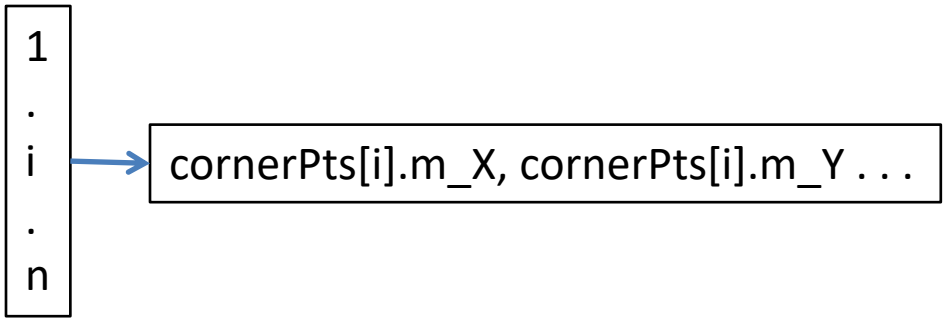
- Part b. void HarrisCornerDetector(QImage image, double sigma, double thres, CIntPt **cornerPts, int &numCornerPts, QImage &imageDisplay)
- To do:

  i. Compute x and y derivatives of the image, use them to produce 3 images (I_x^2, I_y^2, and I_x*I_y) and smooth each of them with the Gaussian
  OR, first apply the Gaussian and then compute the derivatives.

  ii. Compute the Harris matrix H for each pixel.

  iii. Compute corner response function R = Det(H)/Tr(H), and threshold R. Try threshold 50 on the UI.

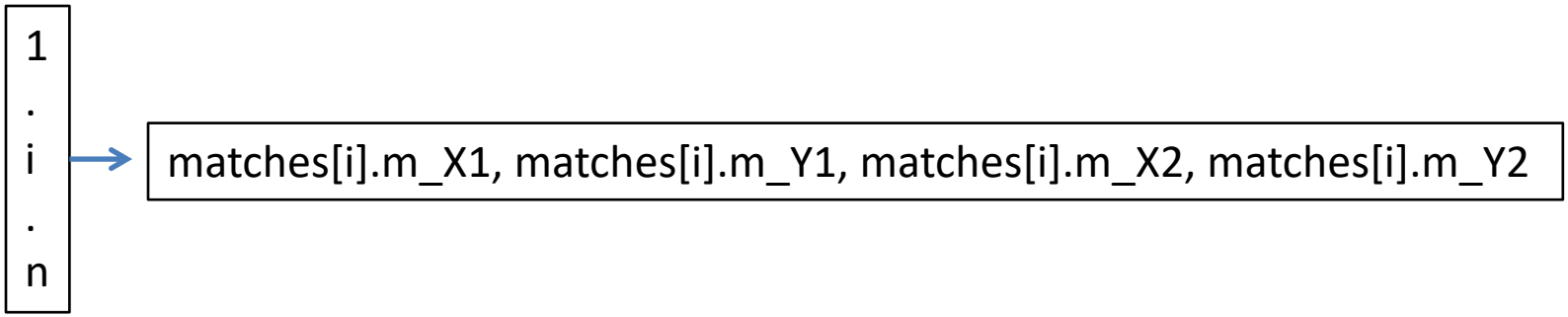
  iv. Find local maxima of the response function using nonmaximum suppression.



Step 2: (10 pts) Implement MatchCornerPoints.

- void MatchCornerPoints(Qimage image1, Cintpt *cornerPts1, int numCornerPts1, Qimage image2, Cintpt *cornerPts2, int numCornerPts2, CMatches **matches, int &numMatches, Qimage &image1Display, Qimage &image2Display)

- **image1** is the first input image
- **image 2** is the second (match from image 1 to image 2)
- **cornerPts1** is a vector of interest points found in image1
- **cornerPts2** is a vector of interest points found in image 2
- **numCornerPts1** is the number of interest points in image1
- **numCornerPts2** is the number of interest points in Image 2
- **matches** is a vector of matches; each match has X and Y coordinates from each image



Step 2: (10 pts) Implement MatchCornerPoints.

- void MatchCornerPoints(Qimage image1, Cintpt *cornerPts1, int numCornerPts1, Qimage image2, Cintpt *cornerPts2, int numCornerPts2, CMatches **matches, int &numMatches, Qimage &image1Display, Qimage &image2Display)

To do this you'll need to follow these steps:

a. Compute the descriptors for each interest point.
This code has already been written for you.

b. For each corner point in image 1, find its best match in image 2.
The best match is defined as the closest distance (L1-norm distance.)

c. Add the pair of matching points to "matches".

d. Display the matches using DrawMatches (code is already written.)
Just pass it the required parameters.
You should see many correct and incorrect matches.

Step 3: (10 pts) Compute the **homography** between the images using **RANSAC** (Szeliski, Section 6.1.4). You write the helper functions for ComputeHomography.

- Part a. void Project(double x1, double y1, doube &x2, double &y2, double h[3][3])
  - This should project point (x1, y1) using the homography "h".
  - Return the projected point (x2, y2).
  - See the slides for details on how to project using homogeneous coordinates.

- Part b. **int** ComputeInlierCount(double h[3][3], Cmatches *matches, int numMatches, double inlierThreshold).
  - This is a helper function for RANSAC to compute thenumber of inlying points for a homography "h".
  - Project the first point in each match using the function "Project".
  - If the projected point is less than the distance "inlierThreshold" from the second point (in that match), it is an inlier.
  - Return the total number of inliers.

c. void RANSAC (Cmatches *matches , int numMatches, int numIterations,
double inlierThreshold, double hom[3][3], double homInv[3][3],
Qimage &image1Display, Qimage &image2Display)

- matches is a set of numMatches matches
- numIterations is the number of times to iterate
- inlierThreshold is a real number so that the distance from a projected point to the match is less than its square
- hom is the homography and homInv its inverse
- Image1Display and Image2Display hold the matches to display

c. void RANSAC (Cmatches *matches , int numMatches, int numIterations,
double inlierThreshold, double hom[3][3], double homInv[3][3],
Qimage &image1Display, Qimage &image2Display)

a. Iteratively do the following for "numIterations" times:
   i. Randomly select 4 pairs of potentially matching points from "matches".
   ii. Compute the homography relating the four selected matches with the function "ComputeHomography. "
   iii. Using the computed homography, compute the number of inliers using "ComputeInlierCount".
   iv. If this homography produces the highest number of inliers, store it as the best homography.

b. i. Given the highest scoring homography, once again find all the inliers.
   ii. Compute a new refined homography using all of the inliers (not just using four points as you did previously.)
   iii. Compute an inverse homography as well (the fourth term of the function ComputeHomography should be false), and return their values in "hom" and "homInv".

c. Display the inlier matches using "DrawMatches".

Step 4: (10 pts) Stitch the images together using the computed homography.
Following these steps:

a. bool BilinearInterpolation(Qimage *image, double x, double y, double rgb[3]).
This code should be the same as in Assignment 1, but if x and y are out of range, it should just return false, else fill the rgb and return true.

b. void Stitch(Qimage image1, Qimage image2, double hom[3][3], double homInv[3][3], Qimage &stitchedImage)

- image1 and image 2 are the input images
- hom is the homography and homInv its inverse
- stitchedImage is the result.

i. Compute the size of "stitchedImage. "
To do this project the four corners of "image2" onto "image1" using Project and "homInv". Allocate the image.

ii. Copy "image1" onto the "stitchedImage" at the right location.

iii. For each pixel in "stitchedImage", project the point onto "image2". If it lies within image2's boundaries, add or blend the pixel's value to "stitchedImage. " When finding the value of image2's pixel use BilinearInterpolation. (Here's where the true/false is needed.)