**Lucas-Kanade  
ECE 847 Assignment #6**

**Purpose:** In this assignment you will implement the Lucas-Kanade algorithm to detect and track features through a video sequence of images.

**Before you start:** Watch the Lecture 6 series.

**Background info:** Just as binocular stereo matching determines the correspondence between two images taken by different cameras, optical flow determines the correspondence between two images taken by the same camera and two different times (typically consecutive frames of a video sequence separated by a fraction of a second). Unlike stereo matching, optical flow is not constrained to lie along the epipolar line, thus necessitating a 2D (rather than a 1D) search for every pixel / feature. A classic algorithm for computing sparse optical flow is Lucas-Kanade, which uses the optical flow constraint equation, combined with the assumption that all pixels in a small neighborhood move at the same velocity, to compute the inter-frame displacement of a feature point. Feature points are detected in the first frame by examining the eigenvalues of the gradient covariance matrix surrounding the pixel. Lucas-Kanade remains popular because it is relatively straightforward to implement, computationally efficient, and generalizes well to more involved problems that require affine or projective motion models.

**Instructions:**

1. In this assignment you will write an application (that is, you will modify the code in *homework*) to detect and track features through a video sequence of images. The code should perform the following steps in order when it is run:
   1. Reads 3 command-line parameters, which we will call *filename-format*, *first-frame*, *last-frame*, *sigma*, and *window-size*.
   2. Use *filename-format* and *first-frame* to *last-frame* to determine the filename of the first image of the sequence. This is easily accomplished using CString::Format, which works like sprintf. For example,  
      CString filename;  
      const char\* format = “img%04d.bmp”; // format string – could be CString instead  
      filename.Format(format, 400); // ‘filename’ is now “img0400.bmp”
   3. Detect good features in the first frame. Remember that your parameters for feature detection do not have to be the same as those for feature tracking. Therefore, you may want to hardcode a different  for your gradient computation here (independent of the *sigma* parameter), and use a small 3x3 window for constructing the gradient covariance matrix (as opposed to using *window-size*). Use the Tomasi-Kanade method of thresholding the minimum eigenvalue to compute a measure of “cornerness” for every pixel in the image. Then perform non-maximal suppression to set the “cornerness” to zero for every pixel that is not a local maximum in a 3x3 neighborhood (using either 4- or 8-neighbors). Note that, unlike Canny, this non-max suppression will not care about the direction in which neighbors lie relative to the pixel, but instead will consider all the pixels in the neighborhood at once. You may also want to enforce a minimum distance between features, but it will be simpler to either allow no more than 1 feature in each 8x8 image block, or to increase the value of  (not *sigma*) accordingly.
   4. Loop over all the frame numbers, from *first-frame* to *last-frame*, using *filename-format* to determine the filename of the current image.
   5. For each pair of consecutive frames, perform Lucas-Kanade tracking of all the features to update their 2D image positions. Use *sigma* to compute the gradient of the image, and use *window-size* as the size of the window over which to accumulate information for constructing Z and e. Keep the feature coordinates as floating point values throughout the tracking process, only rounding for display purposes; to handle non-integer values, use bilinear interpolation. Do not worry at all about declaring features lost, but simply allow them to continue tracking throughout the sequence, even if they drift to a neighboring surface in the world due to occlusion. Nevertheless, be sure to perform bounds checking so that features that reach the image border do not cause the program to crash due to out-of-bounds memory access.
2. It is very much recommended that you create a synthetic pair of images with known motion between them to test your algorithm. To do this, load a real image, shift it to the right by one pixel (for example), then write out the result as a second image. Together, these two images for your synthetic pair with known motion for all pixels as (1,0). Once your code works for this pair, then create another synthetic pair with different motions to continue your testing. Once you are convinced that it works on these pairs, then run your code on a real sequence with unknown motion. Lucas-Kanade is not difficult to implement, but it is difficult to debug due to the fact that it is hard to visualize what the program is doing. Approaching your coding in this systematic manner will make it much easier to verify whether the program is working, as well as to determine where the bugs in your program are.
3. Your output should look like this:
   1. One figure windows should show the first image with the features overlaid. An additional figure should the output of feature tracking over time. That is, as you track the features, display the result in this second figure window, so that the result appears to the user as a movie of results. To overlay the features, set a 3x3 region of pixels around the feature coordinates (rounded to the nearest integer) to the color red. Feel free to display any additional output you desire, but be sure to set the title of each figure to an appropriate human-readable string that indicates what is being displayed.
4. A word about the quality of your results: Keep in mind that you are implementing the simplest version of the algorithm. Do not expect your results to be perfect by any means: Some percentage of your features will not track well, and that is okay. And when a feature is near the boundary of an object, expect it to move in unpredictable ways, which is due to occlusion. Nevertheless, consider your code a success if some percentage of your features appear as though they are latched onto the image, not floating on top of it. If all the features are floating, then you probably have a bug.
5. The grader will test your code on the video sequences in flowergarden.zip and statue\_sequence.zip. The motion in the former is more predictable, and therefore will require a small *sigma*, than the latter.
6. For this assignment, you may ***not*** use any of the Lucas-Kanade or KLT implementations in Blepo, or any other existing implementations of Lucas-Kanade. You also may not use any of the Interp functions.
7. Submit your code to the grader as described in the first assignment.