Notes on the CV.jit Objects

The CV.jit objects are a suite of objects by Jean-Marc Pelletier that perform computer vision operations on jitter matrices. They are available at http://jmpelletier.com/cvjit/.
The same site holds his excellent jit.freenect.grab that brings the power of the Xbox Kinect to jitter.
The only documentation to the cv objects are in the help files. Luckily these are pretty extensive, but you may need to follow a round robin from helpfile to helpfile to figure some things out. The information about how to actually use the objects is best found by exploring some of the abstractions supplied for the helpfiles, which often contain the word draw in the title.

Preparation

Effective computer vision often depends on modifying the image we are looking at to exaggerate specific features. The following objects will be useful, or even required for the processes discussed later.

Black and White

Most of the cv.jit objects will only work on a greyscale image. The easiest way to derive this is with jit.rgb2luma, although is some special cases you may want to split off one color instead. That is done with jit.unpack or a one layer jit.matrix with a planemap attribute.

Resizing

Like any jitter objects, the cv.jit objects are most efficient with small matrices. The cv.jit.resize objet offers better (but slower) interpolation than you get by just running the image through different size matrices. Here are some examples:

Original
Figure 1a


jit.matrix reduced


enlarged again


cv.jit.resize reduced

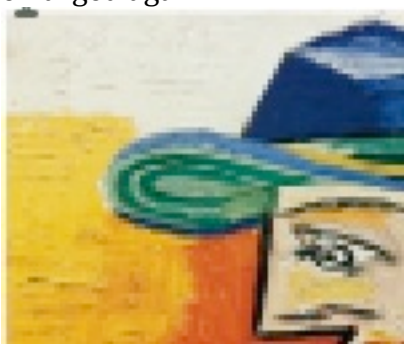

enlarged again

Figure 1b

As you can see in figure 1, the effect is like checking the cubic spline resample box in Photoshop Image Size. Straight edges soften up a bit, but the overall effect is less harsh, and really more accurate.

Threshold

Converting an image into distinct area of black or white is a necessary preparation for many of the cv processes. In the usual technique a jit.op with one of the logic ops such as > is used. However, this is only useable in near perfect situations. With real images the lighting can vary enough across a scene to render this ineffective. Cv.jit.threshold takes different approach. From the cv.jit.threshold helpfile:

> Uneven illumination is often a problem when thresholding an image. Adaptive thresholding addresses this issue by adjusting the threshold based on the brightness of an area surrounding each pixel. The difference between the pixel value and the average brightness of pixels within the distance set by the "radius" attribute is calculated. If it is greater than the "threshold" value, the pixel value is set to ON. If the "mode" attribute is set to 1, the calculation is reversed – in order to select dark regions instead.

Figure 2 shows how well this works. The source image has dots of various darkness. Thresholding with jit.op on the left is very sensitive to the absolute level of the image. The fainter dots are left out. Of course the threshold cold be lowered, but in most real life images, that would also bring up a lot of garbage. In CV.jit.threshold, we are really looking at the local contrast, so if the eye can see it, the patch can probably detect it. Figure 3 shows an even more difficult situation.

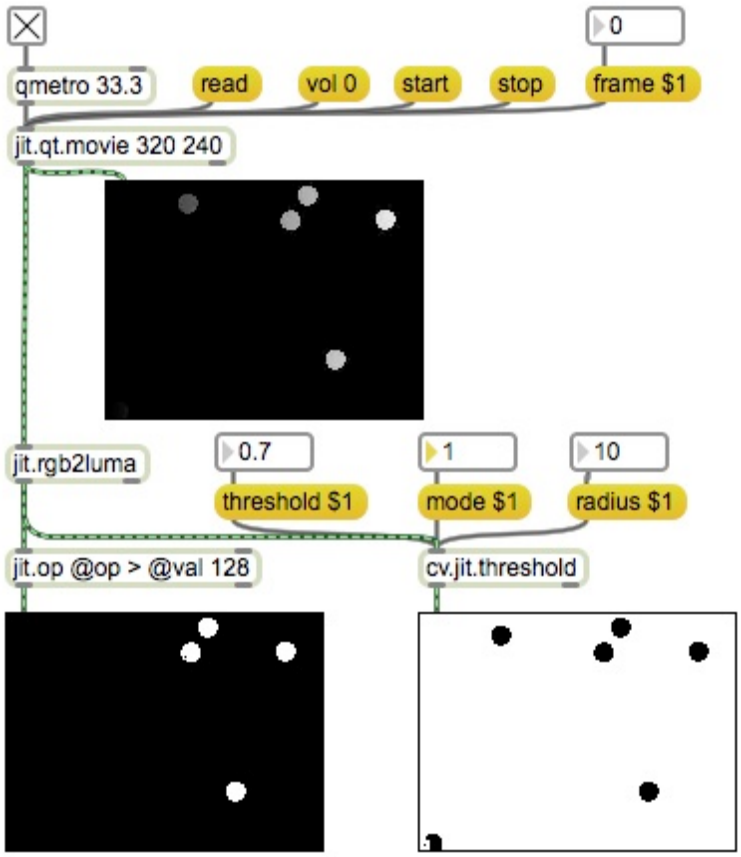
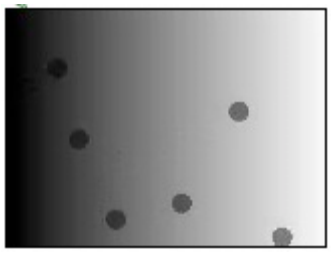
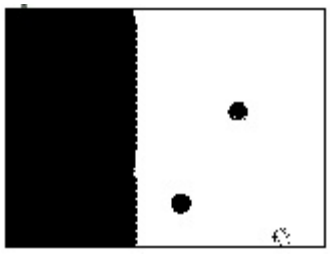


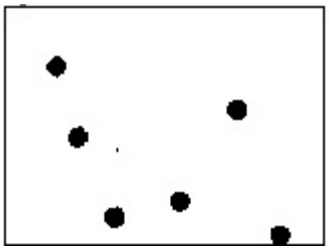
Figure 2. cv.jit.threshold compared to jit.op



original



jit.op



cv.jit.threshold

Figure 3.

Running Average

cv.jit.ravg will compute the average value of each pixel during a scene. If the image is stable, this will essentially be the background for any quickly moving objects. The absolute difference between the running average and current frame will go a long way toward isolating the interesting parts of the scene. Moving objects will leave a ghost, but that may actually be helpful in the thresholding process.

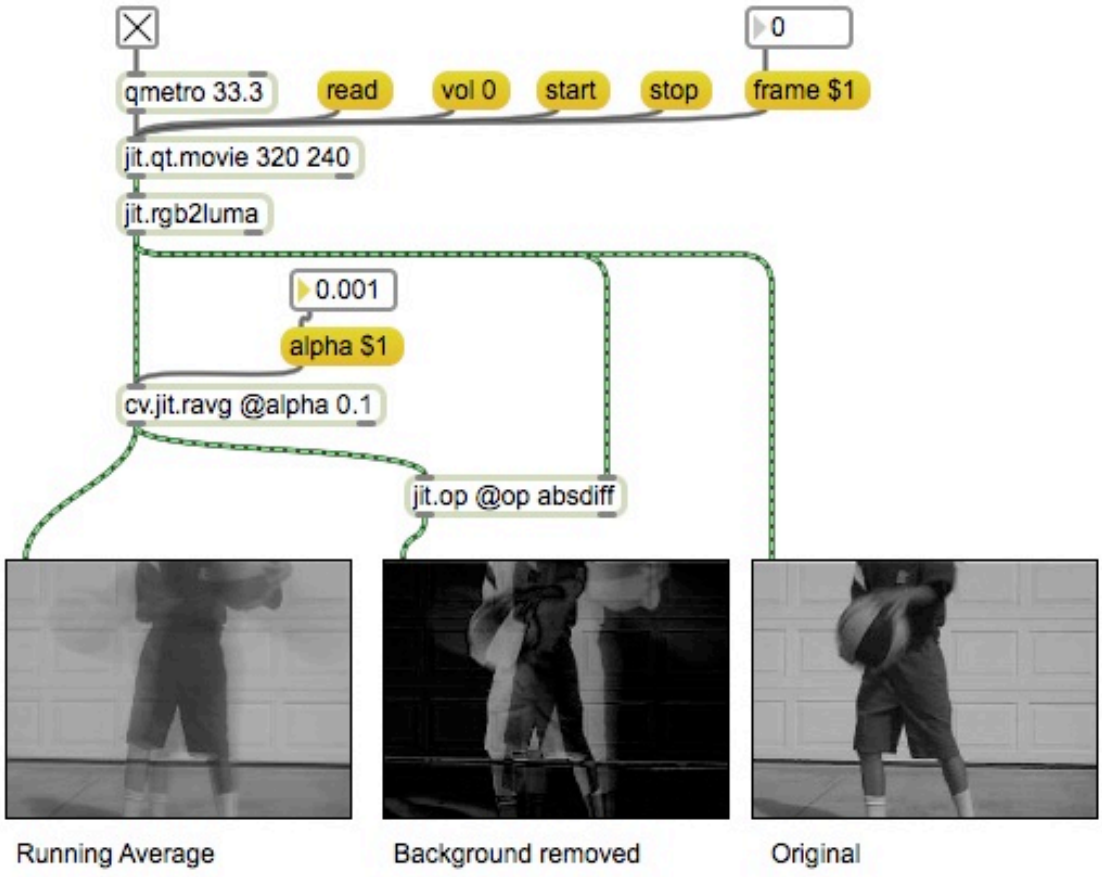


Figure 4

Pattern recognition.

Pattern recognition is a basic aspect of computer vision. The most advanced pattern recognition in the cv library uses the cv.jit.moments object to analyze shapes, and then either cv.jit.undergrad or cv.jit.learn to detect those shapes. An eventual version of this tutorial will include that process. For quick and dirty pattern recognition, the library includes simple tracking an face detection.

Faces

cv.jit.faces will detect a face in an image. That is, the presence of a face, but not a specific face. A face is defined as features similar to two eyes, a nose and a mouth in the correct relative positions. The face also needs to occupy an area of approximately 20x20 pixels. The image needs to be prepared by converting to greyscale and size reduction appropriate to the source material. Jit.rgb2luma produces a greyscale image and cv.jit.resize will accurately reduce the size of the image.

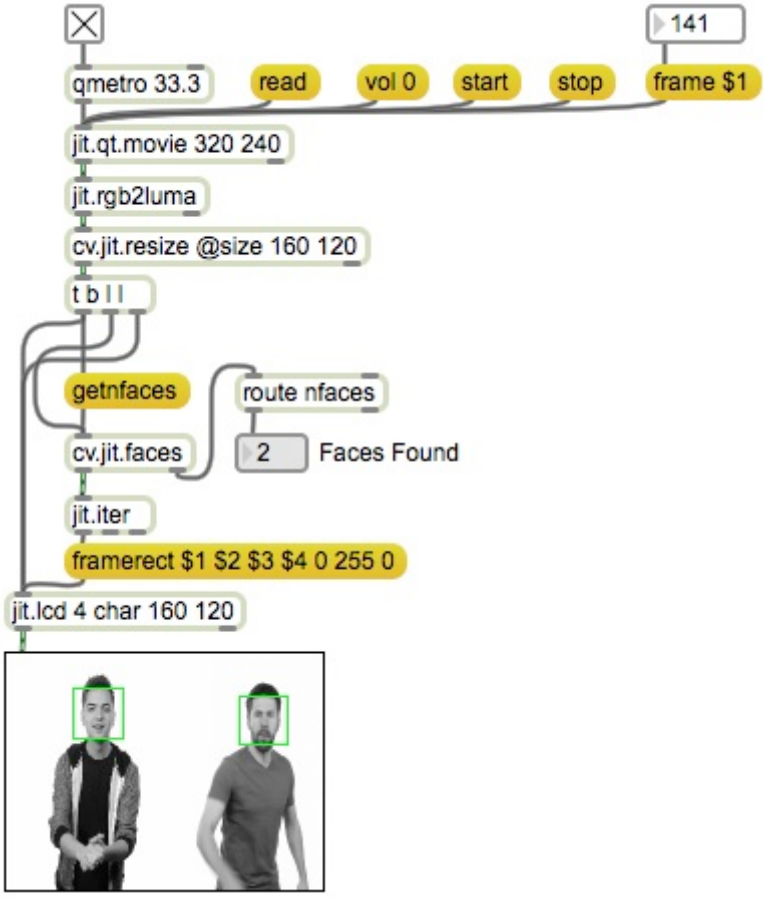


Figure 5.

The output of cv.jit.faces is a matrix containing one cell per face. The contents of the cell frame a square area around the face. The message getnfaces will trigger a report of the number of faces currently detected, but does not go below 1. If no faces are found, the output will be a single cell matrix containing all 0s. The cells, once

separated by jit.iter are properly formatted to draw a box around the face as shown in figure 5, or isolate a face in a window as in figure 6

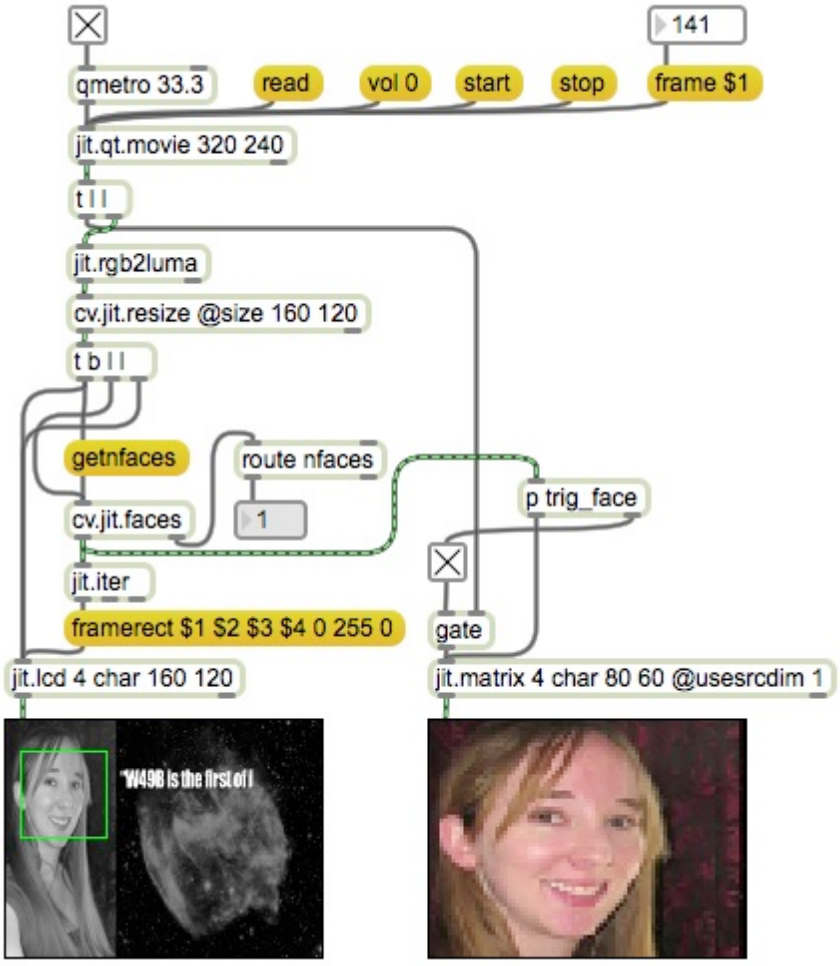


Figure 6a

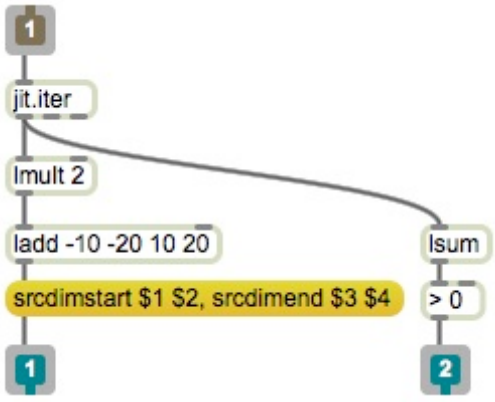


Figure 6b Contents of trig_face

In figure 6b, note the use of source dimension to grab the appropriate area of the original image.

Tracking

cv.jit.track will track an object as it moves around the image. The process requires a greyscale image which is easily produced by jit.rgb2luma. The number of points to track is set by the npoints attribute. There is no limit on this number, but presumably trying to track too many will slow things down. The message [set index X Y] will begin tracking a point at location X Y. The index determines where the point will be reported in the output matrix.

Movement of a point is inferred by "optical flow". That is, the relative intensity of pixels surrounding the point are recorded and matched to nearby areas in the next frame. If the same pattern is found in a nearby location the point is assumed to have moved. The points are output as a 3 layer matrix long enough to hold all points. Jit.iter will split this into lists of X Y and status (1=active, 0 = lost). Figure 3 shows one way to use this data. The Lswap and Ladd objects convert the X Y position into a rectangle that can be used to draw, for instance, a red dot.

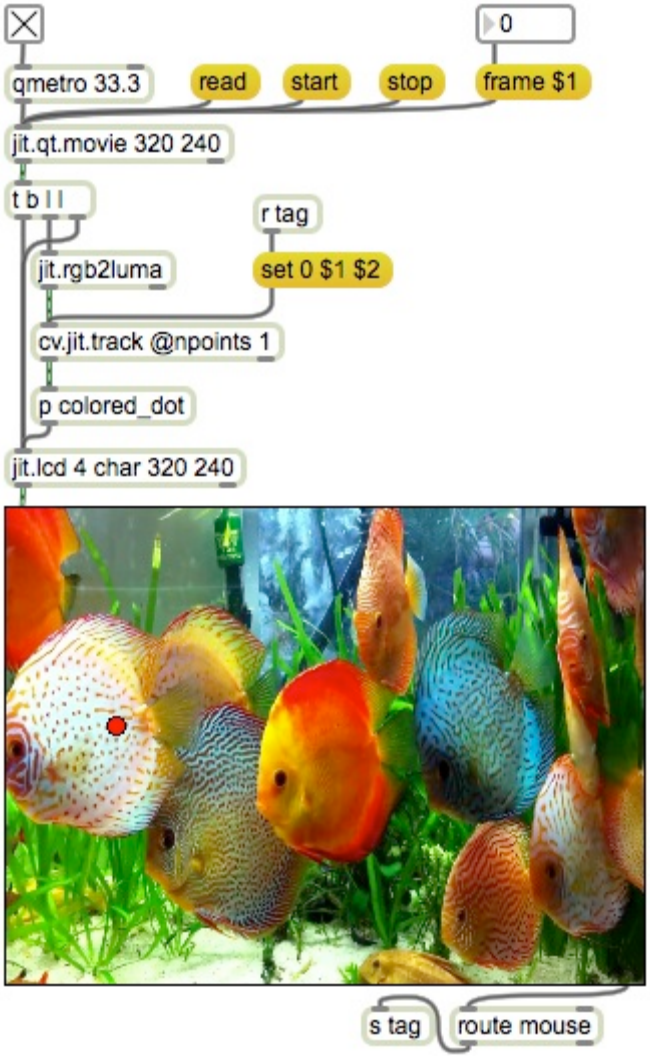


Figure 7. Note red dot on white fish to left. That dot follows the fish.

The point to track is set by clicking on a particular fish. This sends teh mouse message from the jit.pwindow as "tag", which produces a set message to cv.jit.track.

Figure 8 shows how the circle is superimposed on the image. First a trigger (t) object sends the matrix to jit.lcd as the background. Note that the list token (l) is used to get a matrix through trigger. The middle outlet of trigger sends the same image to be processed, first by jit.rgb2luma then to cv.jit.track. after all processing is done (which results in the appearance of the circle) a bang move the new image out of jit.lcd to the window. This is easily modified to color many dots by using the index output of jit.iter to pick the color.

Figure 8.

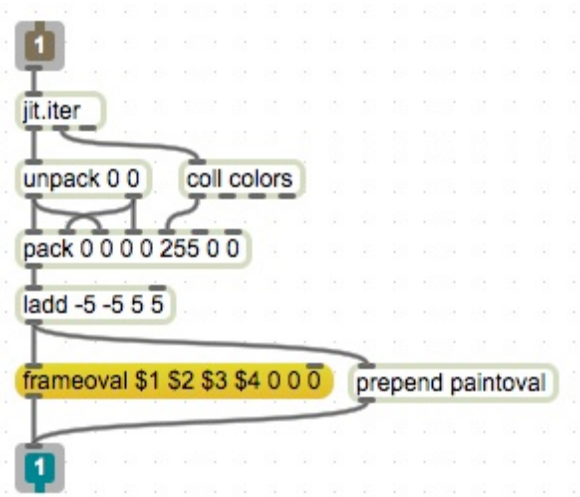


Figure 9.

Blobs

cv.jit.label

There are many applications where we want to identify and track individual shapes Once images have been through threshold to isolate the brightest or darkest areas, cv.jit.label can attach numbers to each distinct region or "blob". The output of label is a one plane matrix of either long or char of the same size as the input. Each cell contains the number of the region it is assigned to.

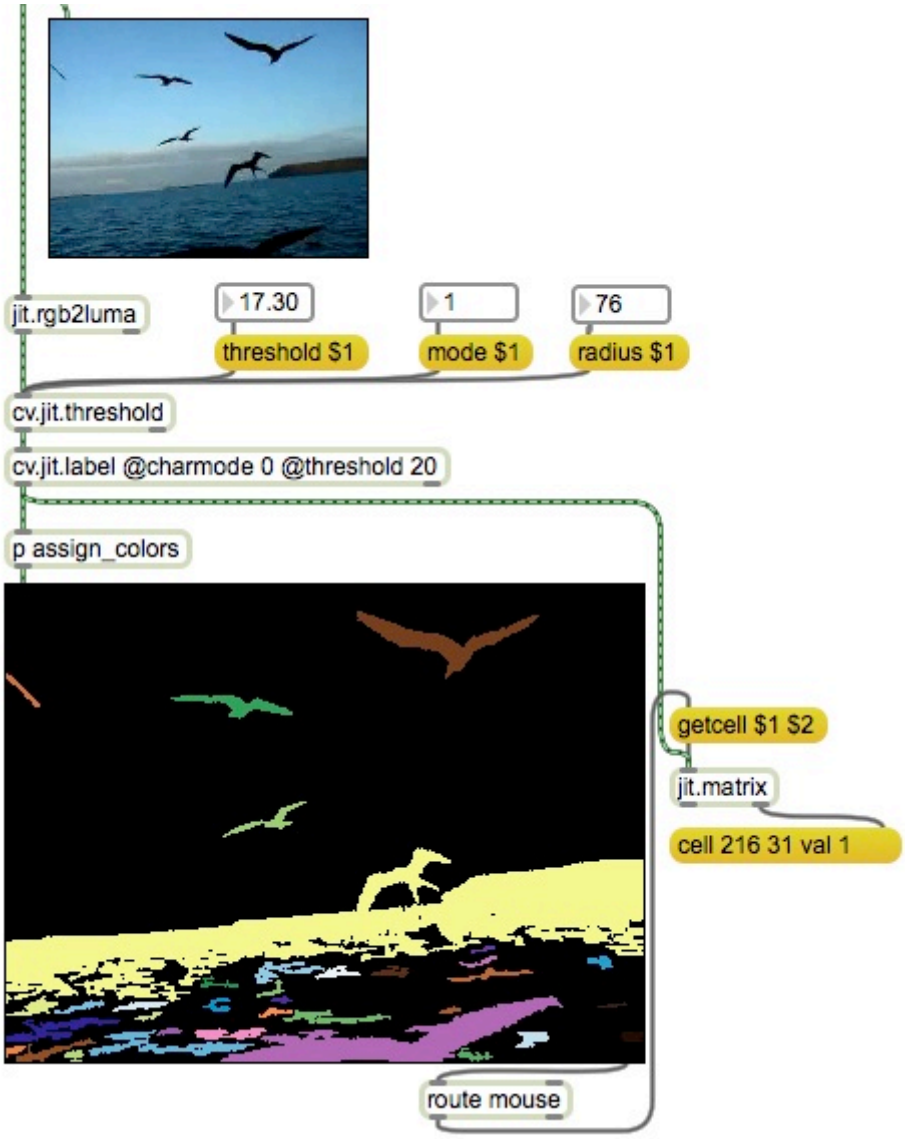


Figure 10.

In figure 10, I have assigned a color to each blob. The region number of any pixel can be found by a mouse click on a blob. One of many possible methods of assigning color is shown in figure 11.Coloring the blobs is only a tutorial aid, what we really want is to find the location of individual blobs. Figure 12 shows one technique.

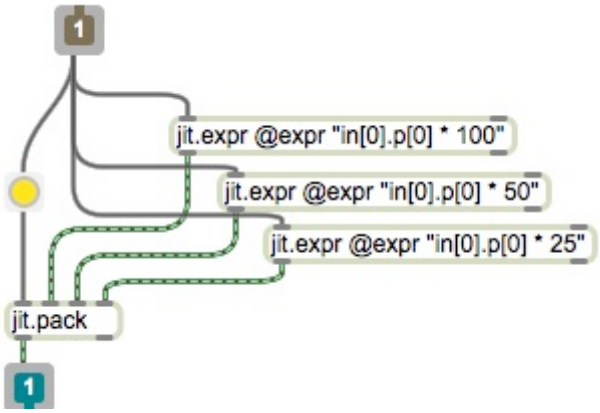


Figure 11.

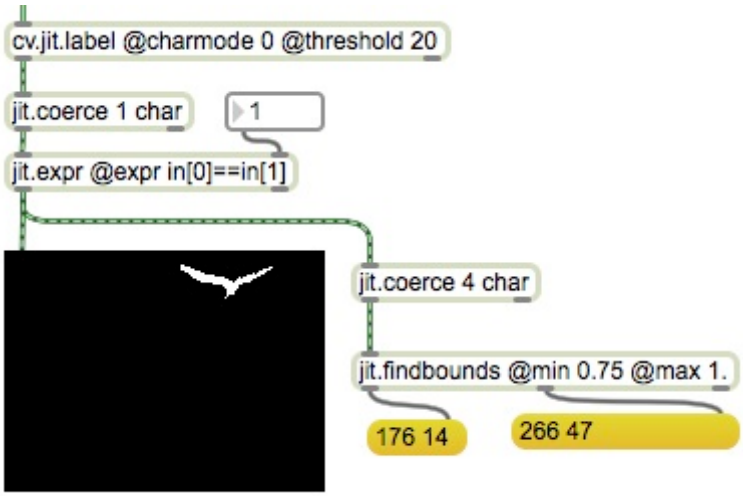


Figure 12.

In Figure 12, the output of cv.jit.label is a matrix of long. In order to be properly manipulated by jit.op or jit.expr, long type matrices must be coerced to char type. This is because the internal operations of both jit.expr and jit.op are done in floating point, but as a convenience to users of the ARGB color system, numbers are mapped from the range 0 255 to 0-1.0. When this is done, inaccuracies in the conversion make it impossible to use the equality operator. When a matrix is coerced, the normalize operation is skipped, so a value of 1 in a long matrix remains a 1 in a char matrix. Once this is done, we can identify which cells are equal to any given index. Figure 12 shows the location of region 1.

cv.jit.blobs.centroids

Once an image is labeled, there are many cv.jit operations available, some quite sophisticated. Several of these operations are in the cv.jit.blobs family. Cv.jit.blobs.centriod will find the center point of each region. Figure 13 shows it in action.

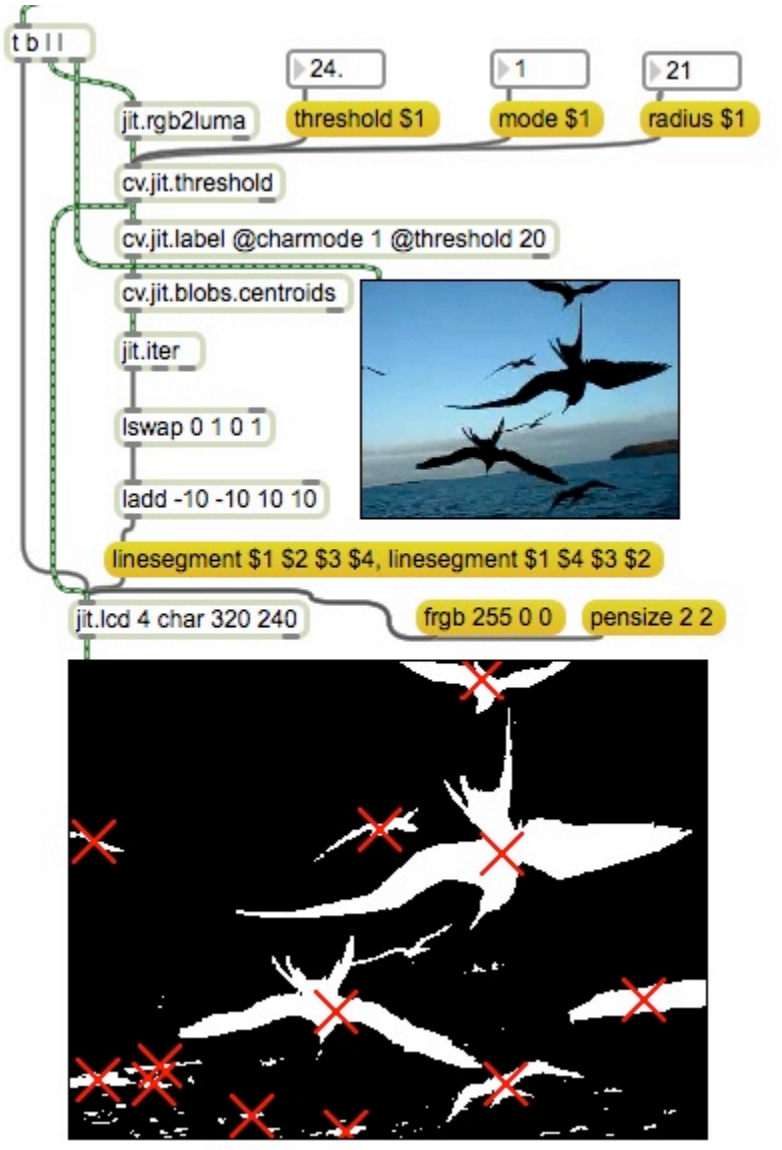


Figure 13
The output of cv.jit.blobs.centroids is a matrix with X Y location and area of each region identified by the threshold and label process. In figure 13, the XY values are used to draw an X over each region. The area values could be used to restrict the Xs to the largest objects, eliminating a lot of the noise from the ocean surface in this image.

One flaw in cv.jit.labels is that the regions are arbitrarily numbered top down and left to right. (That's from the first pixel in each region.) That means if objects trade places, we loose track. Figure 14 illustrates this-- as the large bird moves down the frame, his tracking color changes. He is region 2 in the left image, and 4 in the right. Cv.jit.blobs.sort can help with this, but only in certain situations. Cv.jit.blobs.sort follows the motion of the centroids and reassigns index numbers according to the minimal motion from frame to frame. When blobs overlap, they merge into one. That means one region will disappear from the list.

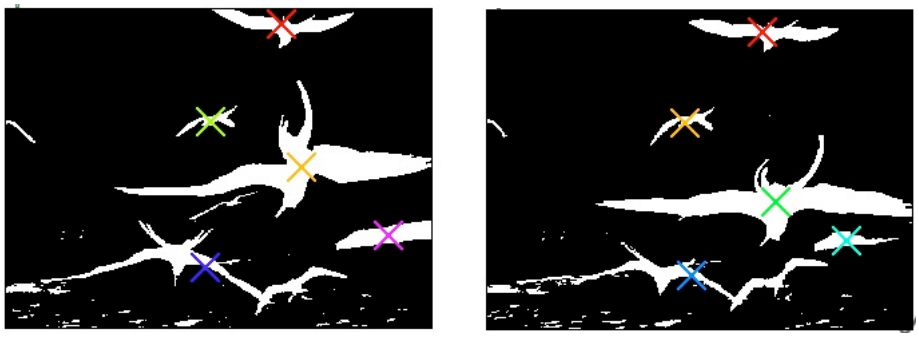


Figure 14

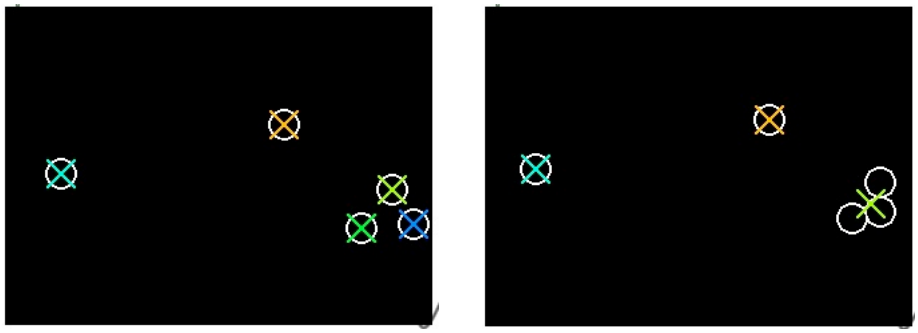


Figure 15

The movie in figure 15 is 5 bouncing dots. In the left frame, the dots are distinct, in the right three have merged. There is no way to prevent confusion in this case.

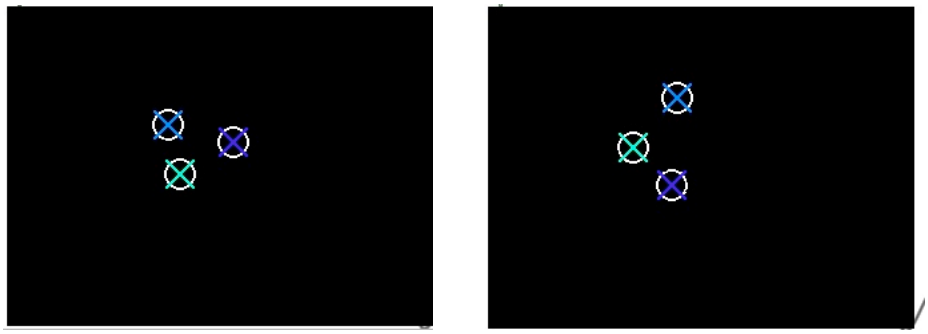


Figure 16.

In figure 16, the situation is simpler, in that the dots don't collide as often. Here you can see cv.jit.blobs.sort has kep the correct colors assigned, even as the objects swap places. The subpatch assigning the colors is shown in figure 17

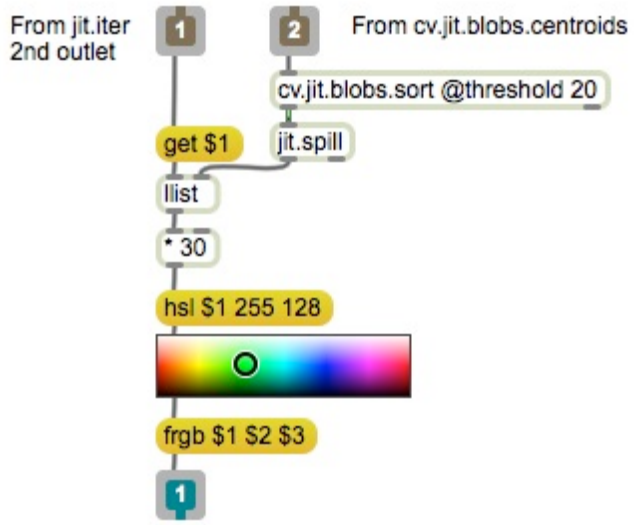


Figure 17. cv.jit.blobs.sort at work.

Some other cv functions

This section is incomplete.

Dilate and erode

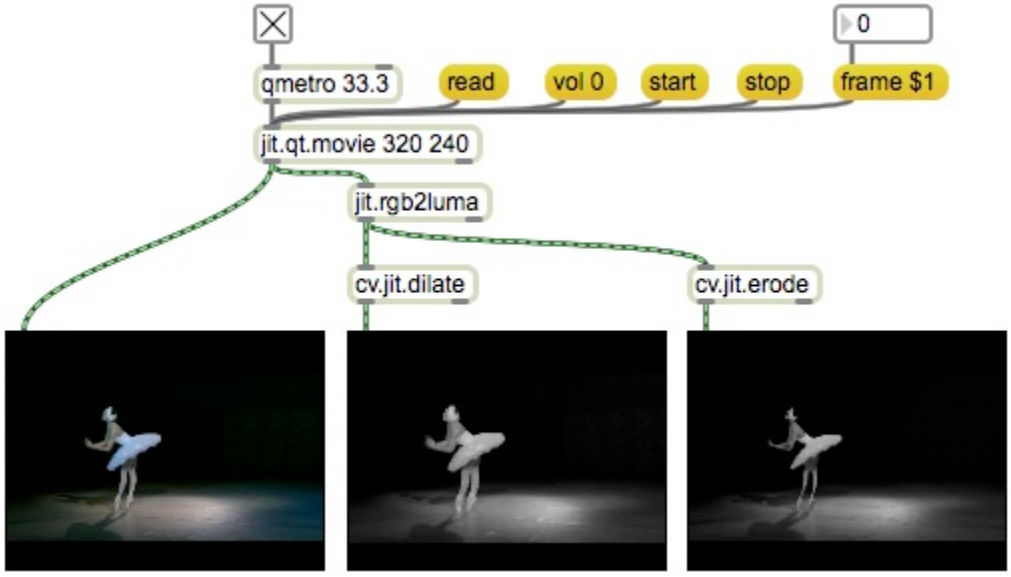


Figure 18
cv.jit.dilate changes the pixel to match the greatest value of any surrounding pixel. Cv.jit.erode changes the pixel value to match the minimum value of the surrounding pixels.

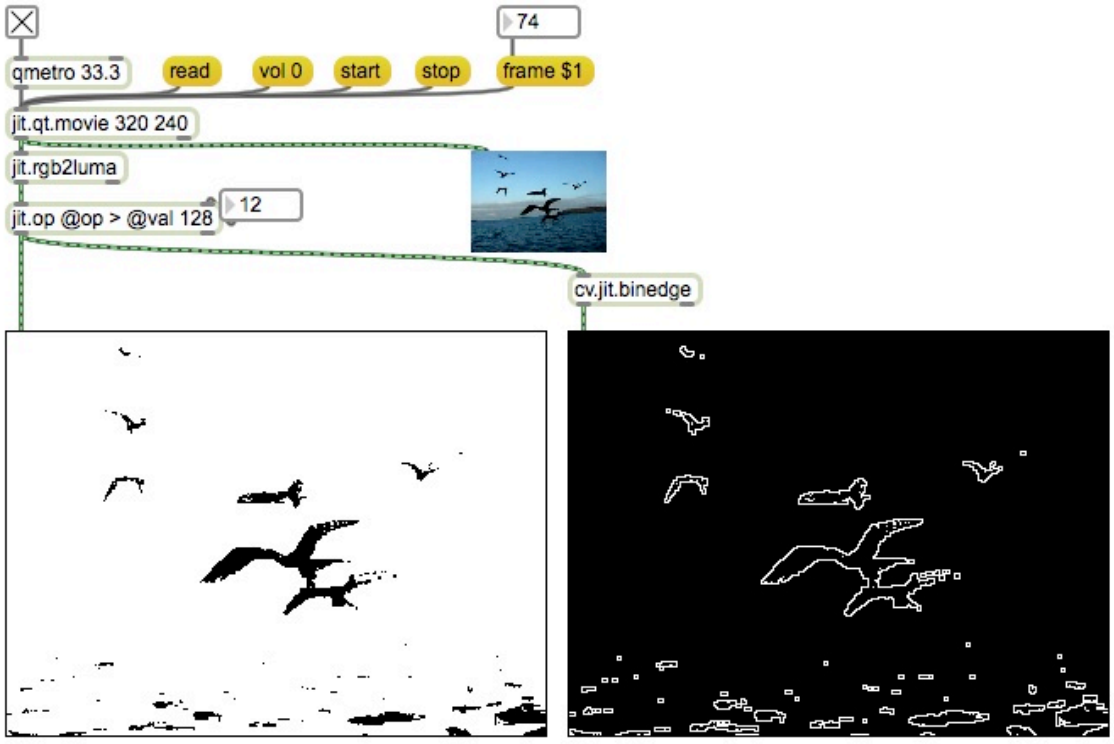


Figure 19 cv.jit.binedge

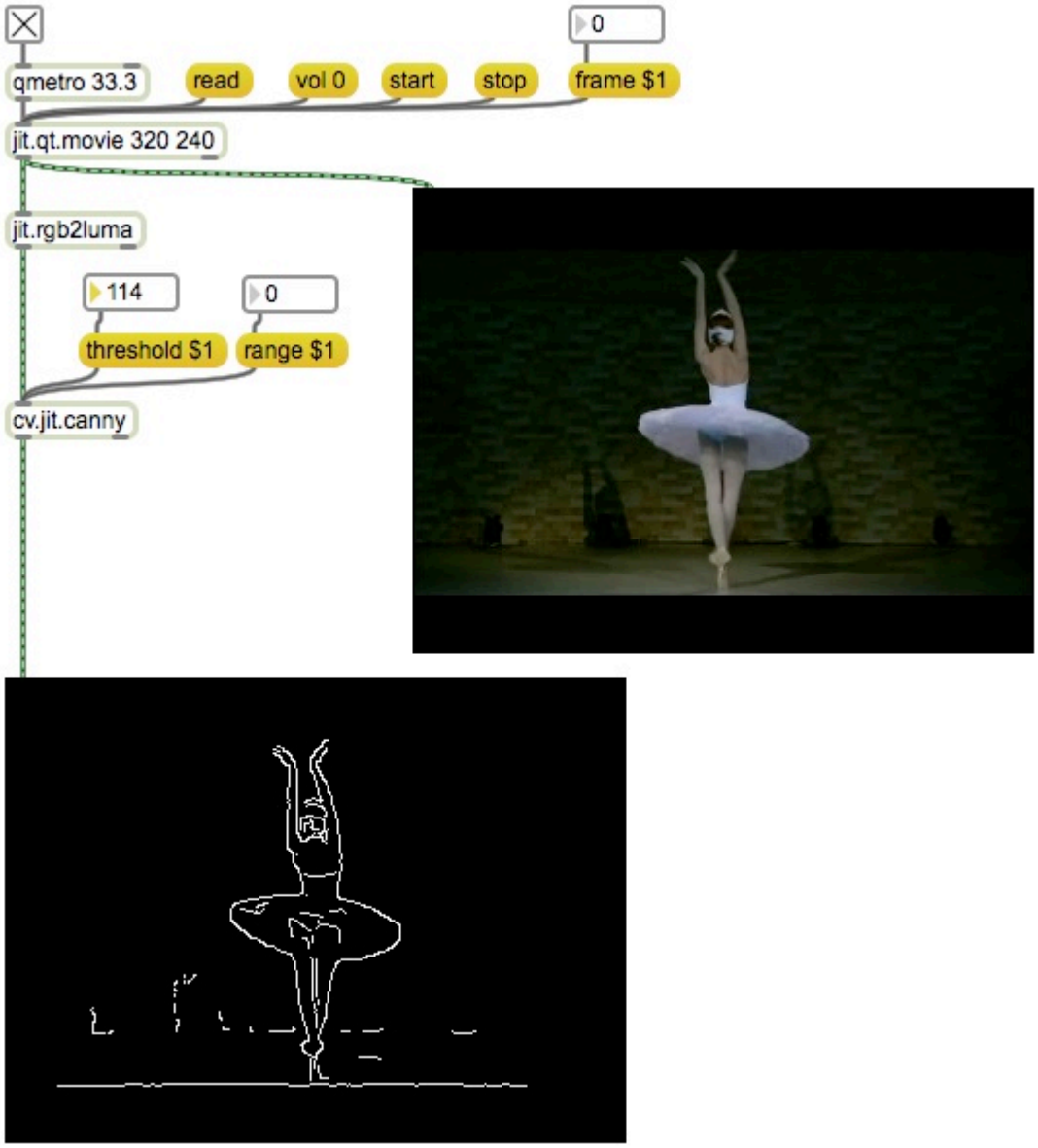


Figure 20. cv.jit.canny (edge detection)