

cv.jit

computer vision externals for jitter

Object Documentation

documentation,
externals,
abstractions,
and patches
by

Jean-Marc Pelletier

jmp@iama.ac.jp

www.iama.ac.jp/~jovan02/

© copyright 2003-2004

Statistics

cv.jit.mean

Computes the mean value of each pixel over a number of frames. Works a lot like the regular Max object “mean”. In order to clear the matrix, you must send the “reset” message rather than “clear”, as “clear” will not reset the internal frame counter to zero. Accepts any type or plane count. Note, however, that due to rounding errors, char and, to a lesser extent, long calculations are going to deviate downwards from the actual mean. If accuracy is an issue, or you plan to feed **cv.jit.mean** a large number of frames, convert to floating point beforehand.

cv.jit.covariance

Computes the covariance matrix of a vector. Accepts only 1-dimensional float32 or float64 data. For n -sized vectors, outputs an $n \times n$ matrix. For every element in the input vector, the mean value of that element is subtracted from the input value. The average of each of these deviations is then calculated. The output matrix contains the product of these means, so that for a 3-element vector with mean deviations (a,b,c) , the covariance matrix will be:

$$\begin{matrix} a*a & a*b & a*c \\ b*a & b*b & b*c \\ c*a & c*b & c*c \end{matrix}$$

Although seemingly arcane, the covariance matrix can have its uses for tasks like pattern recognition.

cv.jit.sum

Adds the value of all the cells in each plane and outputs the result in a list. Accepts any type or plane count.

cv.jit.variance†

Calculates the variance of the input matrices. The variance is defined as the average of the square of the sample values minus the mean.

cv.jit.stddev†

Computes the standard deviation of the incoming matrices. The standard deviation is simply the square root of the variance, so the same result can be obtained with **cv.jit.variance** and a jit.op object. The standard deviation is a measure of how much sample values vary from the mean, or in other words, how wide the distribution on either side of the mean is. About 65% of sample values are within one standard deviation of the mean, whereas 95% are within twice that value. This measurement is very useful when it comes to setting bounds or threshold values, for instance in a background subtraction operation. If the mean value of a background pixel is 50, and the standard deviation is 10, then a pixel valued at 80 would be considered foreground. However, if the standard deviation is around 30, there is a good chance that it belongs to the background.

Motion

Optical Flow

Optical flow is the measure of how much a pixel moves from one frame to another. If we assume that every pixel in a given image was obtained by translating a pixel in the previous frame, then the optical flow of that image is the displacement, relative to the X and Y -axes, of all the pixels. Of course, computing the optical flow isn't all that easy, often there is absolutely no way of knowing which pixel in one image corresponds to which in the previous. This is why the optical flow can only be guessed rather than calculated. Nevertheless, these estimations provide useful information about how each part of the image is moving over time. For the moment, **cv.jit** offers two of the most common optical flow algorithms: the Lucas-Kanade technique, and the Horn-Schunk technique.

cv.jit.LKflow

Estimates the optical flow using the Lucas-Kanade method. This algorithm is based on the assumption that pixels move only a short distance from one frame to another. It has the advantage of yielding good results for slow-moving objects but unfortunately gives completely unreliable results for faster movements. The performance can be greatly improved, both in terms of CPU usage and reliability, by using small matrices. 80×60 matrices, or smaller, gives very good results.

Accepts 1-plane char data. Note that the image is assumed to be greyscale and *not* binary, as the luminosity of the pixels is important to the algorithm. The output is a 2-plane float32 matrix. The first plane contains the horizontal displacement; negative values mean leftward motion, positive, rightward. The second plane contains the vertical data; negative values mean *upward* motion, positive, *downward*.

You can adjust the performance of the algorithm via the "radius" attribute. Values can range between 1 and 7. Higher *radius* values give cleaner, more uniform results but the resulting flow data will appear "blurred" and CPU load will increase significantly. Default value is 2.

cv.jit.HSflow

Estimates the optical flow using the Horn-Schunk method. This technique assumes that changes in optical flow, over the image, are relatively uniform. It is slightly better at estimating faster movements than the Lucas-Kanade technique, but it will also become unreliable if the motion is too fast. It also deals better with large uniform areas than Lucas-Kanade.

Like **cv.jit.LKflow**, accepts 1-plane char data and outputs a 2-plane float32 matrix containing horizontal and vertical flow.

cv.jit.track

Tracks the position of a number of pixels over time. This object uses the pyramidal implementation of the Lucas-Kanade technique described by Jean-Yves Bouguet.

The Lucas-Kanade method of calculating optical flow, as stated above, is very precise for small movements but is incapable of accurately guessing large motions. The algorithm used here circumvents this problem by first guessing the position of the target pixel on a very low-resolution copy of the image ($1/8$ the original size in this implementation.) Even large movements will have only small pixel displacement on such a small matrix. This first guess is

then used to make another guess on a larger copy, and the process is repeated until the new position is guessed on the original image.

This technique is too computationally intensive to be used for every pixel, as in the case of **cv.jit.Lkflow**. However, it is perfect for a small set of target pixels. The maximum number of pixels that can be tracked simultaneously has been set to 255.

Before you can use **cv.jit.track**, you must specify how many points you wish to track using the “npoints” attribute. You can freely change this value later on, to suit your needs. You set the starting position of each point using the “set” message. “set” takes three arguments. The first one is the point index. If you are tracking 12 points, indices will go from 0 to 11; they will range from 0 to 254 if you are using 255 points. The two other arguments are the x and y values for the pixel. You can of course reset the position of any pixel at any time by re-sending a “set” message with the proper index.

The output of **cv.jit.track** is a 3-plane float32 matrix. This matrix will be as wide as the number of points specified by “npoints”, with every column corresponding to one point. The first and second planes contain the estimated x and y position of the pixel. Those values are floats, as the algorithm can only calculate the most likely place for the pixel to be, rather than its exact position. The third plane contains the pixel status. Normally, this will equal 1. However, there are times when the algorithm will declare a pixel “lost”. In this situation, the status cell will equal 0. Pixels usually get lost when they exit the frame. You may choose to ignore this value, or use it to reset the lost point to another position. The algorithm will try to find a lost pixel from its last valid position, which is why a lost pixel can revert back to a “1” status.

cv.jit.trackpoints

When using large number of points with **cv.jit.track**, displaying the position of these points can be problematic. This external only accepts the 3-plane float matrices output from **cv.jit.track**. You must manually specify the size of the output matrix with the “size” attribute, followed by the width and height. Normally, this will be the same as the size of the matrices being sent to **cv.jit.track**. The cells of the output matrix corresponding to the position of the points in the input matrix will be set with each point’s index values. In order to avoid labelling points starting at 0, all indices are offset by one. I.e. 1 to 255, rather than 0 to 254.

cv.jit.trackgroup

This object is used to manage the input and output of **cv.jit.track**. In order to improve on the robustness of a tracking method, it is useful to use a group of pixels, instead of just one, to track an object. That way, if a pixel gets lost or wanders off the object, the other pixels can continue tracking.

The output of a **cv.jit.track** object is connected to the object’s left inlet. It will automatically detect the number of points in the matrix and assume all these points belong to the same group. The right outlet of **cv.jit.trackgroup** should be connected back to the **cv.jit.track** object. You initialize the object by sending a coordinate in the right outlet. When you do so, the object will generate the right number of random points in the vicinity of the pixel specified and output “set” messages to the track object.

The mean position of all the pixels in the group will be calculated and output from the left outlet. **cv.jit.trackgroup** will check to see if all pixels are within a certain distance of this mean value. If a pixel wanders too far away from the centre, it is deemed lost and reset within acceptable range. You can adjust the size of the area used to generate random points and reset lost pixels, through the “dispersion” message. The “tolerance” message, on the other hand sets the distance from which a pixel is considered lost. You must make sure that

the tolerance value should not be too close to, or smaller than, the dispersion. This would result in pixels being reset in a position that's already considered lost!

You can use several **cv.jit.trackgroup** objects to track several points at once. You can connect all these objects' outputs to a single **cv.jit.track** object, provided you do two things: first you must split the output of the track object (using **jit.scissors**) so that each trackgroup only receives the coordinates of the points that belong to its group. Secondly, you must explicitly tell **cv.jit.trackgroup** the offset value of its group's pixels using the "offset" message.

If you wanted to track three points, you would initialize a **cv.jit.track** object with an "npoints" value of, say, 27. This will give you 9 points per group. You can easily change this number by setting a different "npoints" value, though making sure it is a multiple of 3. You then send each **cv.jit.trackgroup** offset values of 0, 9, and 18 respectively. Split the output of **cv.jit.track** with **jit.scissors @columns 3** and plug each of these into the right trackgroup.

Doing this instead of using three **cv.jit.track** objects greatly improves performance. Calculating the scaled-down matrices is one of the heaviest parts of the track algorithm, and by using only one object you avoid going through this step several times.

Binary Images

Binary images are defined as 1-plane char matrices. Every zero valued pixels are said to be OFF, while non-zero pixels are ON. Binary images are extremely useful, not only as masks, but also for such diverse tasks as shape analysis and image segmentation. Unless specified, all the objects described in this section accept and return binary matrices.

Image transformation

Image transformations are usually meant to pre-process your data before using it. The following abstractions can be used to remove noise, or to smooth your data. You can link several of these objects for a more pronounced effect.

cv.jit.binedge

Returns only edge pixels. “Edge” pixels are pixels that are ON and have less than eight ON neighbours. In other words, they are ON pixels that have at least one OFF neighbour.

cv.jit.dilate

In binary mode, a pixel will be marked ON if any of its neighbours is ON. This will make shapes fatter and although it will make noise all the more noticeable, it is a good way to get rid of small holes in an image. In greyscale mode, however, each pixel is given the maximum value of the pixels around it. You can toggle between both mode using the “greyscale”, or “grayscale” attribute followed by a 0, or a 1. You can also change the shape of the neighbourhood with the “mode” attribute. Mode 0 uses 8 neighbours, and mode 1 uses 4 neighbours forming a cross.

cv.jit.erode

This operation does somewhat the opposite of “dilate”. In binary mode, a pixel will stay ON only if *all* of its neighbours are also on. This can be a good way to get rid of noise but will also make your shapes thinner and holes wider. In greyscale mode, a pixel will get the minimum value of its neighbours. “Greyscale” and “mode” attributes function as for **cv.jit.dilate**.

cv.jit.open†

An “open” operation is simply an “erode” followed by a “dilate”. This results in a slight widening of holes and openings, hence the name. It gets rid of smaller specks of noise while retaining the general size of the input objects, unlike “erode” alone.

cv.jit.close†

“Close” is the opposite of “open”, that is, a “dilate” followed by and “erode”. Like “open”, it outputs shapes that have roughly the same size as the input. Small holes and gaps will, however, be filled in.

Image segmentation

Image segmentation is the task of identifying and isolating *connected components* or, more colloquially, blobs. A blob is a region of continuously ON pixels, effectively an island in a sea of OFF.

cv.jit.floodfill

The floodfill algorithm takes a pixel coordinate specified by the “seed” attributes and checks the value of that pixel. If it is OFF, it does nothing and returns a blank matrix. If the pixel is ON, however, it will “flood-fill” the blob this pixel belongs to with ON values and return only that blob. This works exactly like the flood-fill tools in popular paint or image editing software.

cv.jit.label

This algorithm scans through the image and gives each connected component an individual value. If you set the “mode” attribute to its default value of 0, it will paint the top-leftmost blob with ones, and will number blobs incrementally moving right and down. In mode 1, however, it will paint the blobs with the number of pixels in that blob. This can allow you, for instance, to filter only blobs that have sizes between such and such a value. Furthermore, in either mode, you can use the “threshold” attribute to erase all the blobs that are smaller than the threshold value. This is a very powerful (and surprisingly cheap) way of filtering noise out. In order to accommodate potentially large numbers of blobs, or large blob sizes, the output is a 1-plane *long* matrix. There is a hard-coded limit of 2048 possible blobs.

cv.jit.label now also supports *char* output through the “charmode” attribute. When “charmode” is set to 1 and “mode” is set to the default 0, **cv.jit.label** works just like with long output. If there are more than 255 blobs, only the first 255 encountered will be labelled – other areas being marked 0.

“Mode 1” works quite differently in “charmode”. Blobs are sorted by size and labelled according to their rank. The largest connected component in the image will be labelled with the value 1. The second-largest, with the number 2, and so forth, up to 255.

Image analysis

cv.jit.mass

This external computes the overall “weight” of a greyscale or binary image. For binary data, this amounts to counting the number of ON pixels. Otherwise, the values of all the cells in a matrix are added together. If the data type is *char*, the result is normalised by 255.

cv.jit.centroids

Centroids, or centre of mass, are the coordinates of the point where the weight of pixels left of the x value is equal the weight of pixels to the right, and the weight of pixels over the y value equals the weight below. If there is only a single object in an image, centroids are a very cheap and robust way of doing motion tracking. Note that the centroids do not necessarily fall on an ON pixel in binary images, for instance in the case of a U-shaped object. Since the mass is used to calculate the centroids, **cv.jit.centroids** will also return this value from its second outlet.

cv.jit.moments

This is the most complicated and versatile object of this distribution. Apart from returning centroids and mass, it outputs two lists of shape descriptors.

Moment-based shape analysis is based on the physics concept of *moment of inertia*. Since, in theory, these *moments* are unique to each shape, they have been used for a long time to perform tasks such as optical character recognition.

From the left-most outlet a list of seven *moments* comes out. These are cryptically labelled “*m20, m02, m22, m21, m12, m30, m03*”. The *m* here simply stands for *moment*, the two numbers following tell us how the particular value was calculated.

For every ON pixel, the *x* and *y* indexes are raised to a given power and multiplied together. The numbers following the *m* stand for the powers used for this calculation. For example, an ON pixel at coordinates (5,4) would yield an *m02* value of: $5^0 * 4^2 = 1 * 16 = 16$. The *m21* value would be: $5^2 * 4^1 = 25 * 4 = 100$. Those values are summed up for the whole image, and this result is normalized using the mass. Note also that all coordinates are relative to the centroids, which means that the same shape will return the same *moment* values regardless of where it is. Normalizing also ensures that scale doesn't affect the result either.

Simply by looking at the values we can find out some information about the shape we're analysing:

m20 and *m02* are always positive. Higher *m20* values means that a shape is wider than it is tall, whereas *m02* will be higher if the shape is taller. Note that this is a measure of “weight distribution”, not of absolute size. If there is a single pixel hovering high above a very wide shape, it will not greatly affect *m20* and *m02* values.

m11, m21, and m12 are measures of *covariance*. What this means is that shapes that are strongly diagonal, or skewed, will give higher values.

m30 and *m03* are measures of asymmetry. If *m30* is positive, it means that the shape is bulkier to the left and more outstretched to the right of the centroid. *m03* is similar but for the *y-axis*. If *m03* is negative the shape is more outstretched *upwards* and *downwards* if it is positive. For example, a “thumbs up” shape will have a negative *m03*.

Moments are very useful but they are very sensitive to orientation. In many cases this is not a problem, after all, “thumbs up” doesn't mean the same thing as “thumbs down”. There are situations, however, where this becomes problematic. Imagine for instance, a set-up where the camera is looking at a scene from above. For this reason, **cv.jit.moments** also outputs a list of seven descriptors that are invariant to scale, translation *and* rotation. Those shape descriptors are often called *Hu moments*, after the researcher Ming-Kei Hu.

Hu moments are harder to correlate with obvious features like regular *moments*, which is why their main use is as raw data for shape recognition algorithms.

cv.jit.orientation†

This abstraction accepts the output from **cv.jit.moments'** leftmost outlet and computes the main axis of the shape analysed. This is of course, a more meaningful measurement for narrow objects. The value returned is in radians.

cv.jit.circularity†

This abstraction takes for input a binary image and calculates how “compact” it is. Shapes like squares and circles that are concentrated will return values close to 1. (In theory, a

perfect circle has a circularity of 1.) The closer a shape's circularity is to 0, the more its outline is complicated.

cv.jit.elongation†

This abstraction takes the seven moments output by **cv.jit.moments** and uses them to measure how thin or wide a shape is. Line-like shapes will return low values. As a shape's pixels become more spread out from the main axis, the elongation value increases.

cv.jit.perimeter†

This abstraction simply counts the number of pixels in a shape's edge.

cv.jit.undergrad†

This abstraction performs a simple sort of pattern recognition. You must feed it a number of lists in its right outlet. The lists may represent anything (they can be moments, for instance) but they must all be examples of the pattern you seek to recognise. Once you've trained the object with enough positive data, you can send lists to its left inlet for identification.

cv.jit.undergrad uses a fairly simplistic statistical method for pattern analysis. It looks at the data sent in its right, “training”, inlet and figures out the mean values of each element and how much the values vary from this mean. It then looks at the “candidate” data and assesses whether each element is within acceptable bounds of the mean. If it is, the score is increased by one. For a seven-element list, an output of 7 means a perfect score. An output of 2 would mean that only 2 elements were within identification bounds, while 5 were outside.

You can save and load your training data using the “save” and “load” messages. Clear the object's memory using “clear”.

The quality of the results depends greatly on your training data. If all your lists are very similar the bounds will be set very “tight”. This means that you stand a good chance of running into false negatives. If your training data shows too much variation on the other hand, you risk getting many false positives.

cv.jit.shapeinfo†

This abstraction uses **cv.jit.moments** to derive more shape descriptors that are easy to understand and usable *as is*. Here is a list of the values it computes, from left to right:

Mass: same as above.

Centroids: same as above.

Orientation: the general angle of the shape in radians. Note that the algorithm has trouble differentiating 0° and 45° angles.

Perimeter: the number of border pixels.

Compactness: as the name implies, how compact a shape is. Squares and circles are very compact and as such give *low* values. Intricate shapes give higher values.

Height to width ratio: this value is rotation-independent. Very elongated shapes will have high values, a square will have a value of 1. A 2 by 1 rectangle, a value of 2.

(†) denotes an abstraction