Counting jelly beans: voxel carving and segmentation of a container of heterogenous objects

Alex Cope, Mark O'Meara

March 2014

Abstract

In this report, we discuss our implementation of an ensemble of computer vision methods that serve to help provide an estimate of the number of jellybeans in a glass jar, a well-known parlor game. To do this, we decompose our methods into two spheres: 1. volumetric stereo voxel carving of the jar's convex hull (to estimate the total volume), and 2. segmentation algorithms to count the number of jellybeans visible against the glass wall. Although we did not succeed at ultimately integrating the two prongs of our approach into a final estimate, we did achieve success in each of the two areas of focus, correctly finding the 3D convex hull, and correctly segregating the heterogeneous visible jellybeans.

1 Introduction

The challenge of estimating at a glance the number of jellybeans in a transparent jar has been a classic parlor game for many years. Even as recently as last November, prizes as large as $125,000 have been offered at carnivals for accurate counts of immense numbers of colored beans in gigantic containers.

Despite being somewhat of a toy example, the problem still contains inherent difficulties that we believe are particularly conducive to computer vision techniques. Successful estimation of the total jellybean count requires:

1. An accurate estimation of the size of the containing jar.

2. Proper segmentation of the variegated and heterogeneous jellybeans visible against the glass wall.

In addition, the problem is relatively tractable: the ground truth jellybean count provides a simple measure of the accuracy of any computer vision estimate given. For our project, we implemented an ensemble of computer vision methods to provide estimates for both of these problems. The three-dimensional volume estimation proved particularly conducive to volumetric stereo techniques, particularly an implementation of the voxel carving algorithm discussed in class. And, the jellybean segmentation problem lent itself to a variety of segmentation implementations, notably the watershed algorithm and the mean-shift algorithm.

2 Prior Works

The volumetric stereo methods we used to calculate the container's convex hull are well-researched and tested [1]. The space-carving algorithm that we used operates most successfully if the object does not contain any concavities; our purely convex jar fits this requirement.

In order for voxel carving to perform properly, the world coordinates of each of the camera positions needs to be known via camera calibration. We used the Camera Calibration Toolbox for Matlab, a tried and true software package for generating camera matrices from photographs of a planar calibration target [2].

Finally, for guidance in the general voxel carving procedure, we used the report of Rob Hess from Oregon State University [3].

There exists little research that we could find which sought to count heterogenous objects in a single image (perhaps because of the apparent triviality of the problem); most work done looks at counting objects that move over time, such as pedestrians in a crowd [4]. We relied heavily on the mean-shift algorithm, which was first proposed in 1975 by Fukunaga and Hostetler [5].

3 Approach

3.1 Voxel Carving

In order to estimate the number of jellybeans in our jar, we decided to use volumetric stereo methods to first compute an estimate of the volume of the jar itself; specifically, we used a binary (not RGB) analogue of the voxel coloring algorithm discussed in class, performed on binarized jar images taken by a calibrated camera volume with the background removed. On a binary image, this algorithm is effectively the voxel space-carving algorithm to find the jar's convex hull, but with the *lambda* threshold from voxel coloring. Then, together with a separate calculation of the average size of the jellybean "unit cell" (a voxel-like block representing the average amount of space each jellybean takes up, including the surrounding air), the total count can be found by simply dividing the volume of the jar by the volume of the average jellybean cell.

In order to perform voxel carving, we first had to acquire several different images of the jar taken from different locations; then, the images had to be calibrated, yielding the intrinsic camera parameters, and extrinsic parameters for each image. For reference, we followed the general procedure outlined by [3]. A more detailed discussion of our methods follows:

3.1.1 Data Acquisition

Voxel carving requires that calibrated camera matrices be known for each image. To effect this, we placed a planar calibration target on a table evenly lit by diffuse sunlight, and captured 23 separate images of the target, from positions that formed a camera volume hull above and around the table. For each image of the target taken, we took second photo of the jar on top of the target, placed as exactly as possible in each image. This way, we could reuse the camera matrices found for the images in the voxel carving algorithm.

Figure 1: Example image taken of the jar and calibration target.

It was important that the camera volume lie completely above the jar, in order to satisfy the *ordinal visibility constraint* [1]. This ensures that no scene point falls within the convex hull of the camera centers, which in turn ensures the correct behavior of what the carving algorithm counts as occluded points. See Fig. 1 for an example of one of the images acquired.

3.1.2 Calibration

To calibrate our images, we used the Camera Calibration Toolbox for Matlab package [2]. This package works by gathering a hand-picked origin point on the calibration target, and measures for each of the other corners of the grid (including a grid-spacing measure; ours was 24mm per side). Using gradient descent and successive recalibration on the results, this package helped us to yield a proper intrinsic camera matrix K, as well as extrinsic parameters for each of the 23 cameras in our camera volume, which taken together yielded 23 full camera matrices M. The reprojection error of these camera matrices back onto the target was sufficiently small, and centered around the true world origin (which we placed at the bottom-left-hand corner of the calibration target.) See Fig. 3 for the reprojection errors and visualization of the camera volume in relation to the target.

3.1.3 Voxel Carving

Once the cameras were correctly calibrated, we could implemented the voxel carving algorithm to get an estimate of the jar volume, taken from the convex hull it returns.

Before we could begin traversing the voxels of the scene, however, we had to remove the backgrounds from a subset of the images using Adobe Photoshop, setting each to pure green. In this way, we simulated the green/blue chroma key backgrounds usually used in volumetric stereo implementations. Not needing to use all 23 images to avoid expensive computational redundancy, we chose a subset of 5 images that were spaced evenly around

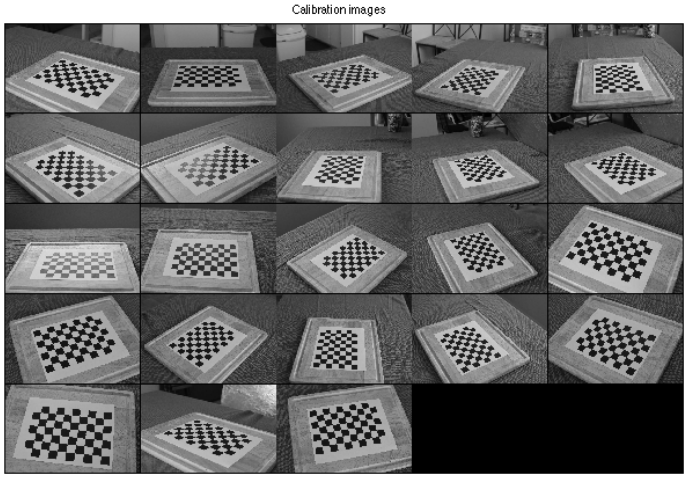


Figure 2: Calibration array of all 23 images.

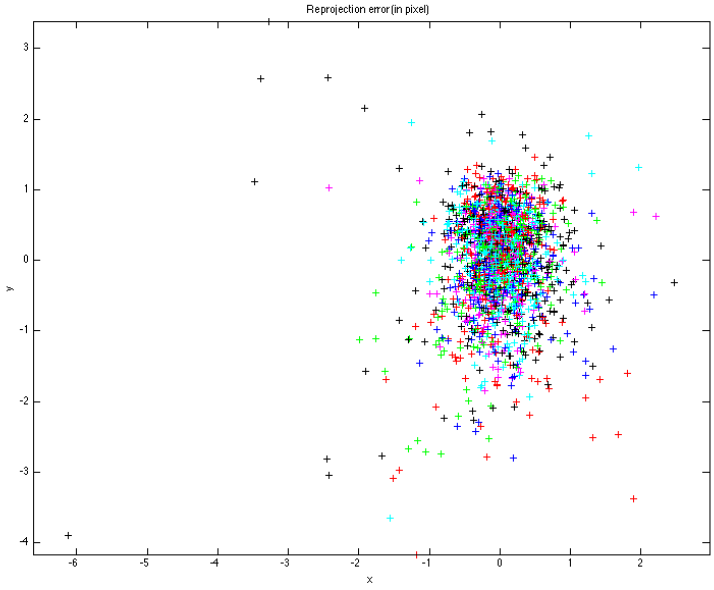


Figure 3: Reprojection error of the images.

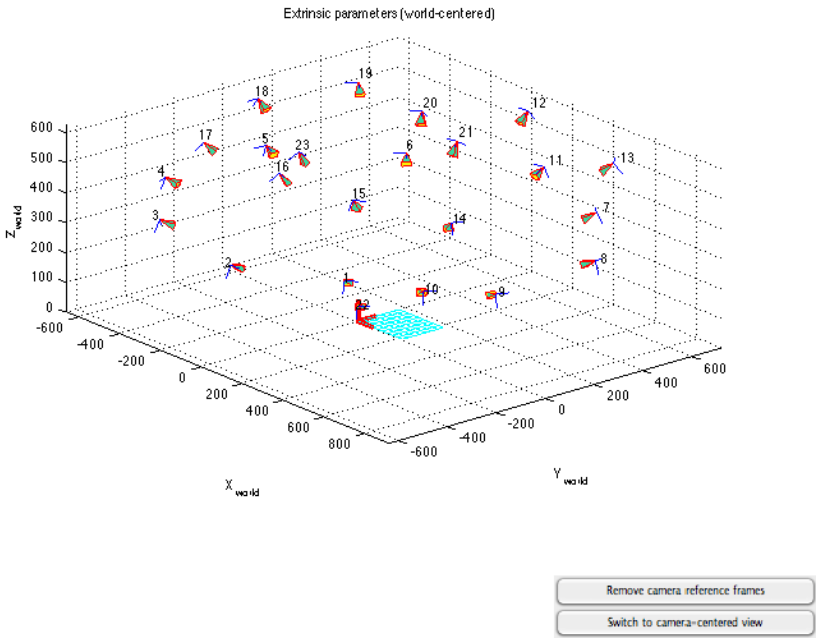


Figure 4: External world parameters for all 23 calibration images.

the jar, yielding a 360-degree view. After this background removal, we transformed the RGB images into binary images, changing the background to white and using a very high threshold of .95 to render the jar as almost completely black. We decided to do this because the reflective, refractive, and highly non-Lambertian surface of the glass jar (as well as the somewhat inconsistent illumination between images) was rendering prohibitive our attempts at true voxel coloring.

After background removal and image binarization, we performed the algorithm proper. To implement the algorithm, we generalized the voxel coloring algorithm implemented in Problem Set 3 to three dimensions. From the input, we take the edge length, in mm, of the voxel size to be measured. Then, we started traversing each voxel in the scene once, starting from the top layer and working layer by layer down to the bottom. Per the ordinal visibility constraint outlined above, it is essential that voxel traversal be carried out top to bottom; this way, voxels chosen in the top layer are properly understood as occluding the voxels behind them in each image, which ensures that the occlusion maps function correctly.

For each voxel traversed, we calculated its projection onto each of the images by applying the proper camera matrix found in the calibration step. This step was a touch more complex than its analogue in Problem Set 3: as opposed to the simple line segments generated in the one-dimensional images there, the result of voxel projections onto 2D images were irregular hexagons. To simply the ability to later index into these projections, we estimated these hexagons using a bounding box around each one. The coordinates of each bounding box were put into a cell array, the elements corresponding to each image's projection.

This projection array was then analyzed for photoconsistency: a background threshold was heuristically chosen to be .1 (this value seemed to work best). If the mean value for the given voxel's projection pixels fell within this threshold of 1 (the pure white background) for *any* of the images, it was removed from consideration and not counted as part of the jar. Otherwise, the pixel values for each camera's projection pixels not present in the images' occlusion maps were put into an array, and the variance was calculated. If the variance was less than the threshold value given by the user (which needs to vary with the voxel size, as shown in PS3), then the voxel was counted as being part of the jar's convex hull. The occlusion arrays for each image were updated, the bits for each projection bounding box being set to 1.

3.2 Segmentation

We tried multiple approaches for solving the segmentation problem. The difficulty with using segmentation to count the number of objects in an image is that the algorithm must be robust to both false positives and false negatives; the count must be exact. The greater the number of objects in the scene, the more difficult this becomes.

3.2.1 Watershed

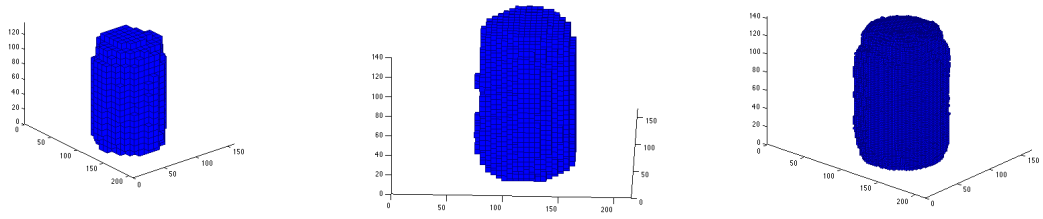
We tried a couple classic blob detectors to count the number of beans and found limited success. The watershed algorithm, developed by Lindeberg (1993), detects blobs using local extrema in the image space [6]. We treat the grayscale image as a topographic surface, with lighter areas corresponding to maxima. Intuitively, we imagine flooding the topographic surface from its minima, and treat each resulting basin as a separate cluster. To reduce over-segmentation due to noise, we can flood the topographic surface from pre-defined markers. OpenCV contains an implementation of watershed but requires the markers to be passed in by the user; we wrote a short algorithm using OpenCV functions like adaptive thresholding to isolate foreground regions of the image to use as markers. Ultimately (see below) this performed at a sub-par level and served as a baseline to the mean-shift approach.

3.2.2 Mean-shift

Intuitively, mean-shift [7] seeks the modes of a feature space. That is, given a certain distribution in some feature space, mean-shift seeks the local maxima of the density of the distribution. Mean-shift associates a kernel window around each data point, then computes the mean of the data within the window. It then shifts the window to its mean and repeats until the window location converges.

More precisely, we initialize one window at each pixel location. Next, we perform mean shift at each window until convergence. The mean shift vector for a given point in feature-space x is given by:

$$m(x) = \frac{\sum_i g(\frac{x-x_i}{h})x_i}{\sum_i g(\frac{x-x_i}{h})} - x$$



(a) 8mm, 8 images (b) 5mm, 5 images (c) 2 mm, 5 images

Figure 5: Convex hull results for varying voxel sizes.

Where $g(x) = k'(x)$, i.e. g is the derivative of the kernel profile of our choosing. We selected to use the Gaussian kernel, which is the standard approach when using mean-shift. h is a user-defined parameter which determines the bandwidth of the kernel window. The smaller h is, the larger the window

We define convergence as when the total sum of all mean-shift vectors is less than a given ϵ, i.e.

$$\sum_i m(x_i) < \epsilon$$

In practice, we found $\epsilon = 1$ performed well without taking too many iterations to reach. To find the clusters, we simply count the number of unique rounded values of one dimension of the final clustered feature set.

We used a five-dimension feature space consisting of the coordinates and RGB values of each pixel. To reduce cluster noise and speed up convergence, we bucketed the feature values into 10 buckets for each dimension (Instead of using the 255 values for each color space dimension).

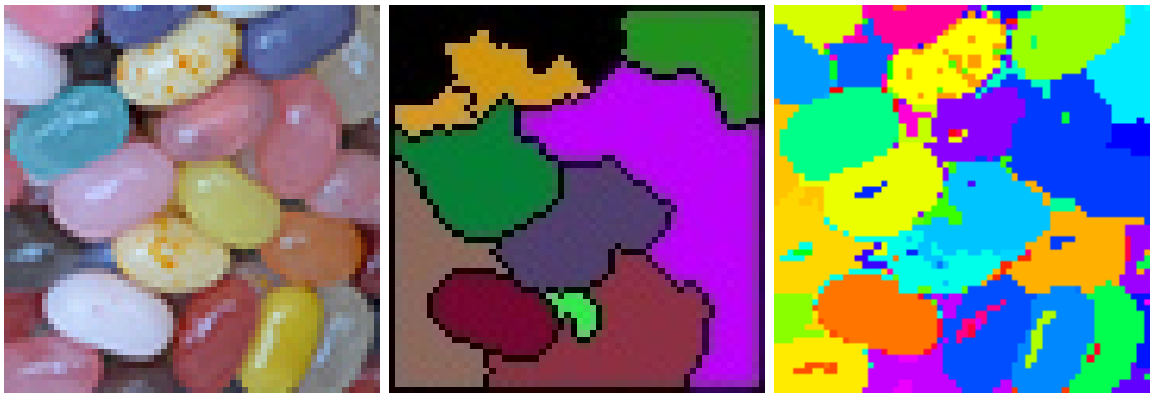
4 Experiments

4.1 Voxel Carving

We used the PATCH3Darray function by Adam Aitkenhead [8] to generate 3D plots of the convex hulls from the voxels that were counted as part of the jar. See Fig. 5 for examples of the jar's shape found for varying voxel sizes (in mm). The algorithm we implemented seemed to work successfully, generating, faithful representations of the three-dimensional shape from all angles. From these hulls, the jar's volume was estimated to be 956.8 cm^3. The jar's actual volume, as measured by us, is 890 cm^3.

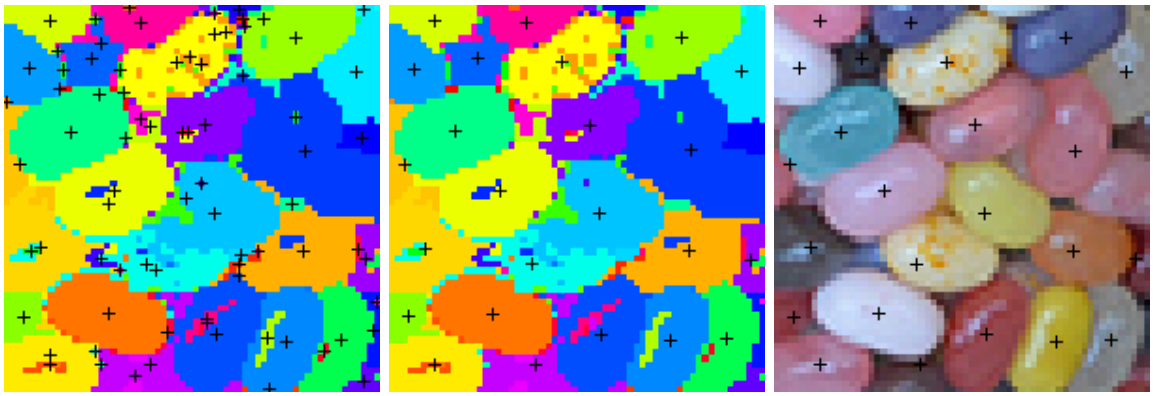
4.2 Segmentation

Different segmentation approaches were testing on the same small (60x60) cropped image of jellybeans (Fig. 6a). A cropped image was used to speed up runtime while iterating through



(a) Unaltered. (b) Watershed. (c) Mean-shift.

Figure 6: Unaltered photo (a). Watershed (b) and mean shift (c) segmentation results. Different colors represent different clusters. (The black lines in the watershed picture are for clarity; they are omitted in the mean-shift picture due to the number of small clusters.)



(a) All clusters. (b) Filtered clusters. (c) Over image.

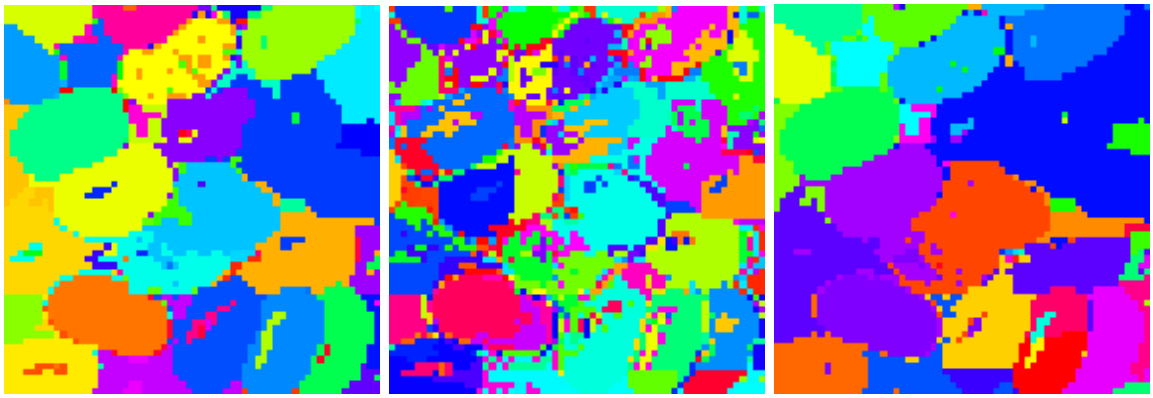
Figure 7: Cluster centroids found using mean-shift.

refinements; this was necessary for mean-shift, which runs in $O(n^2)$ and has two user defined parameters that drastically affect the final result.

Watershed (Fig. 6b) performs poorly compared to mean-shift; it is able to correctly identify one jellybean (in maroon) and otherwise grossly underestimates the number of clusters.

Mean-shift (Fig. 6c) performs much better. However, we can see from the image that it is susceptible to noise and texture (for example, see the number of small clusters it detects on the speckled jellybean at the top) and thus overestimates the number of clusters.

However, we can mitigate this overestimate by culling the clusters using a simple heuristic. We calculate the centroid, area, and variance of each cluster detected by the mean-shift algorithm. If the area is below and the variance above user-set thresholds, then the cluster is ignored. We see the results in Fig. 7. Fig. 7a shows all of the cluster centroids detected by mean-shift ($N = 78$), and Fig. 7b shows only the clusters with area > 25 and variance < 1000 ($N = 24$). These threshold values are obviously affected by the scale of the image; however, because the noisy clusters are all small and all of the jellybeans are roughly the same size,



(a) $h = .8$ (b) $h = .5$ (c) $h = 1.2$

Figure 8: Mean-shift results using different values of h.

the threshold values can be estimated roughly (i.e. they do not have to be fine-tuned for good performance.)

Quantitatively, we see that this cluster detection algorithm performs remarkably well (Fig. 7c). It is able to correctly count all but one jellybean on the front layer, and even correctly identifies some of the occluded jellybeans below the front layer.

One difficulty with mean-shift is that the two user-defined parameters (h and bucket count) drastically affect the final cluster output. There is no automatic way to tune these; one must simply inspect and tune them based on the output. If h is too small, the algorithm will be too sensitive and it will detect too many clusters; also, the algorithm will take much longer to converge. On the other hand, if h is too large, the algorithm will incorrectly merge distinct clusters.

We see this demonstrated in Fig. 8. Fig. 8a-c show the algorithm run with $h = 0.8$, 0.5, and 1.2 respectively. We see that with low h, clusters are more susceptible to noise and more likely to bisect jellybeans. With high h, multiple jellybeans are more likely to be clustered together.

5 Conclusion

In summary, we found limited success in merging our two approaches to the jellybean counting problem together into a single estimate. Although a successful and reasonably accurate measure of convex hull volume was found from our volumetric stereo branch, and despite a measure of success at segregating and counting the jellybeans against the surface of the jar, we could not find a clear way to bring our results together into a final guess.

We suspect this lack of success in integration may be a result of the difficulty of calculating the volume of the jellybean "unit cell" discussed above, which must both include an estimate of the average jellybean volume, as well as the air around it that it takes up. We believe that a successful implementation of voxel *coloring*, as opposed to voxel carving, may have allowed us to estimate this jellybean unit cell to be the largest possible voxel size that yields a reasonable convex hull (because then each voxel would correspond roughly to each jellybean; any larger and the different colors of each jellybean would blend together, severely reducing

photoconsistency). However, the reflective, non-Lambertian surface of the glass jar stymied our efforts at voxel coloring, forcing us to binarize the images in our space carving algorithm, abandoning the color data that the jellybeans provide.

References

[1] Steven Seitz and Charles Dyer, *Photorealistic scene reconstruction by voxel coloring* IEE CVPR, p. 1067-1073, 1997.

[2] Jean-Yves Bouguet, *Camera Calibration Toolbox for Matlab* http://www.vision.caltech.edu/bouguetj/calib_doc, Dec 2013.

[3] Robert Hess, *Adventures in Voxel Coloring* Term Project, University of Oregon. March 2005.

[4] Dan Kong, Doug Gray and Hai Tao *Counting pedestrians in crowds using viewpoint invariant training* BMVC, British Machine Vision Association, 2005.

[5] K. Fukunaga and L. Hostetler *The estimation of the gradient of a density function, with applications in pattern recognition* IEEE Transactions on Information Theory, p. 32-40, 1975.

[6] Serge Beucher, *Image segmentation and mathematical morphology.* http://cmm.ensmp.fr/ beucher/wtshed.html, May 2010.

[7] Dorin Comaniciu and Peter Meer *Mean Shift: A Robust Approach Toward Feature Space Analysis*, IEEE Transactions on Pattern Analysis and Machine Intelligence, p.603-619, 2002.

[8] Adam Aikenhead, *Plot a 3D array using patch*, MATLAB function. Aug 2010.