Contents lists available at ScienceDirect

Computers & Operations Research

journal homepage: www.elsevier.com/locate/cor

Computer image processing

1School of Computer, Wuhan University, Wuhan 430072, China
2Department of computer science, Huazhong Normal University, Wuhan 430079, spain
(2013)

Abstract

The number of bits per screen pixel determines the display's *screen bit depth*. The screen bit depth determines the *screen color resolution*, which is how many distinct colors the display can produce.

Most computer displays use 8, 16, or 24 bits per screen pixel. Depending on your system, you might be able to choose the screen bit depth you want to use. In general, 24-bit display mode produces the best results. If you need to use a lower screen bit depth, 16-bit is generally preferable to 8-bit. However, keep in mind that a 16-bit display has certain limitations, such as

- An image might have finer gradations of color than a 16-bit display can represent. If a color is unavailable, MATLAB® uses the closest approximation.
- There are only 32 shades of gray available. If you are working primarily with grayscale images, you might get better display results using 8-bit display mode, which provides up to 256 shades of gray.

To determine the bit depth of your system's screen, enter this command at the MATLAB prompt.

```
get(0,'ScreenDepth')
ans =
```

The integer MATLAB returns represents the number of bits per screen pixel:

Value	**Screen Bit Depth**
8	8-bit displays support 256 colors. An 8-bit display can produce any of the colors available on a 24-bit display, but only 256 distinct colors can appear at one time. (There are 256 shades of gray available, but if all 256 shades of gray are used, they take up all the available color slots.)
16	16-bit displays usually use 5 bits for each color component, resulting in 32 (i.e., 2^5) levels each of red, green, and blue. This supports 32,768 (i.e., 2^{15}) distinct colors (of which 32 are shades of gray). Some systems use the extra bit to increase the number of levels of green that can be displayed. In this case, the number of different colors supported by a 16-bit display is actually 64,536 (i.e. 2^{16}).
24	24-bit displays use 8 bits for each of the three color components, resulting in 256 (i.e., 2^8) levels each of red, green, and blue. This supports 16,777,216 (i.e., 2^{24}) different colors. (Of these colors, 256 are shades of gray. Shades of gray occur where R=G=B.) The 16 million possible colors supported by 24-bit display can render a lifelike image.
32	32-bit displays use 24 bits to store color information and use the remaining 8 bits to store transparency data (alpha channel). For information about how MATLAB supports the alpha channel, see the section "Transparency" in the MATLAB 3-D Visualization documentation.

Regardless of the number of colors your system can display, MATLAB can store and process images with very high bit depths: 2^{24} colors for `uint8` RGB images, 2^{48} colors for `uint16` RGB images, and 2^{159} for `double` RGB images. These images are displayed best on systems with 24-bit color, but usually look fine on 16-bit systems as well. (For additional information about how MATLAB handles color, see the graphics documentation.) For information about reducing the number of colors used by an image, see "Reducing the Number of Colors in an Image" on page 14-4.

Reducing the Number of Colors in an Image

In this section...

Reducing Colors Using Color Approximation

On systems with 24-bit color displays, truecolor images can display up to 16,777,216 (i.e., 2^{24}) colors. On systems with lower screen bit depths, truecolor images are still displayed reasonably well, because MATLAB® automatically uses color approximation and dithering if needed. Color approximation is the process by which the software chooses replacement colors in the event that direct matches cannot be found.

Indexed images, however, might cause problems if they have a large number of colors. In general, you should limit indexed images to 256 colors for the following reasons:

- On systems with 8-bit display, indexed images with more than 256 colors will need to be dithered or mapped and, therefore, might not display well.
- On some platforms, colormaps cannot exceed 256 entries.
- If an indexed image has more than 256 colors, MATLAB cannot store the image data in a `uint8` array, but generally uses an array of class `double` instead, making the storage size of the image much larger (each pixel uses 64 bits).
- Most image file formats limit indexed images to 256 colors. If you write an indexed image with more than 256 colors (using `imwrite`) to a format that does not support more than 256 colors, you will receive an error.

To reduce the number of colors in an image, use the `rgb2ind` function. This function converts a truecolor image to an indexed image, reducing the number of colors in the process. `rgb2ind` provides the following methods for approximating the colors in the original image:

- Quantization (described in “Quantization” on page 14-5)
  - Uniform quantization
  - Minimum variance quantization
- Colormap mapping (described in “Colormap Mapping” on page 14-9)

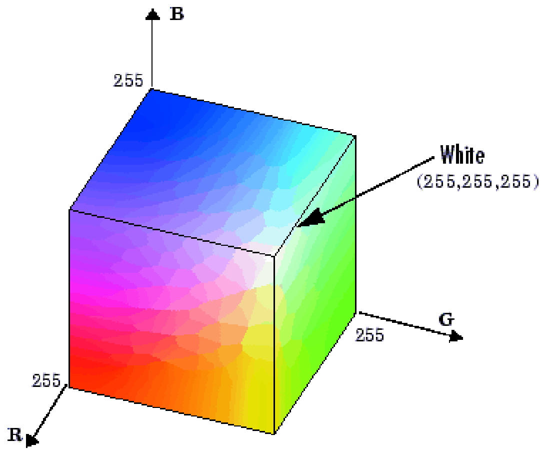
The quality of the resulting image depends on the approximation method you use, the range of colors in the input image, and whether or not you use dithering. Note that different methods work better for different images. See “Dithering” on page 14-11 for a description of dithering and how to enable or disable it.

Quantization

Reducing the number of colors in an image involves *quantization*. The function `rgb2ind` uses quantization as part of its color reduction algorithm. `rgb2ind` supports two quantization methods: *uniform quantization* and *minimum variance quantization*.

An important term in discussions of image quantization is *RGB color cube*, which is used frequently throughout this section. The RGB color cube is a three-dimensional array of all of the colors that are defined for a particular data type. Since RGB images in MATLAB can be of type `uint8`, `uint16`, or `double`, three possible color cube definitions exist. For example, if an RGB image is of class `uint8`, 256 values are defined for each color plane (red, blue, and green), and, in total, there will be 2^{24} (or 16,777,216) colors defined by the color cube. This color cube is the same for all `uint8` RGB images, regardless of which colors they actually use.

The `uint8`, `uint16`, and `double` color cubes all have the same range of colors. In other words, the brightest red in a `uint8` RGB image appears the same as the brightest red in a `double` RGB image. The difference is that the `double` RGB color cube has many more shades of red (and many more shades of all colors). The following figure shows an RGB color cube for a `uint8` image.



RGB Color Cube for uint8 Images

Quantization involves dividing the RGB color cube into a number of smaller boxes, and then mapping all colors that fall within each box to the color value at the *center* of that box.

Uniform quantization and minimum variance quantization differ in the approach used to divide up the RGB color cube. With uniform quantization, the color cube is cut up into equal-sized boxes (smaller cubes). With minimum variance quantization, the color cube is cut up into boxes (not necessarily cubes) of different sizes; the sizes of the boxes depend on how the colors are distributed in the image.

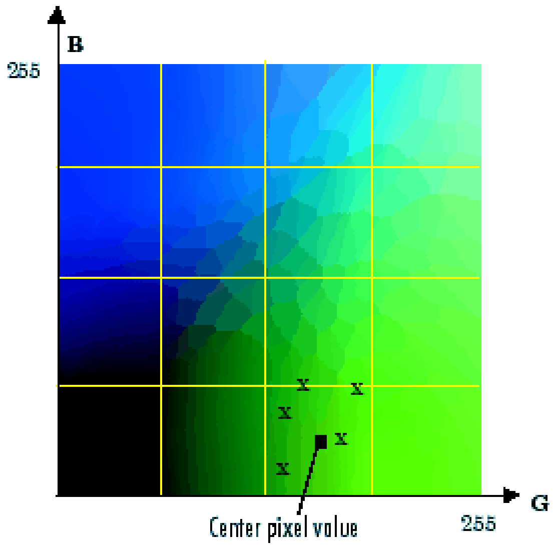
Uniform Quantization. To perform uniform quantization, call `rgb2ind` and specify a *tolerance*. The tolerance determines the size of the cube-shaped boxes into which the RGB color cube is divided. The allowable range for a tolerance setting is [0,1]. For example, if you specify a tolerance of `0.1`, the edges of the boxes are one-tenth the length of the RGB color cube and the maximum total number of boxes is

```
n = (floor(1/tol)+1)^3
```

The commands below perform uniform quantization with a tolerance of 0.1.

```
RGB = imread('peppers.png');
[x,map] = rgb2ind(RGB, 0.1);
```

The following figure illustrates uniform quantization of a `uint8` image. For clarity, the figure shows a two-dimensional slice (or color plane) from the color cube where red=0 and green and blue range from 0 to 255. The actual pixel values are denoted by the centers of the x's.



Uniform Quantization on a Slice of the RGB Color Cube

After the color cube has been divided, all empty boxes are thrown out. Therefore, only one of the boxes is used to produce a color for the colormap. As shown earlier, the maximum length of a colormap created by uniform quantization can be predicted, but the colormap can be smaller than the prediction because `rgb2ind` removes any colors that do not appear in the input image.

Minimum Variance Quantization. To perform minimum variance quantization, call `rgb2ind` and specify the maximum number of colors in the output image's colormap. The number you specify determines the number of boxes into which the RGB color cube is divided. These commands use minimum variance quantization to create an indexed image with 185 colors.

```
RGB = imread('peppers.png');
[X,map] = rgb2ind(RGB,185);
```

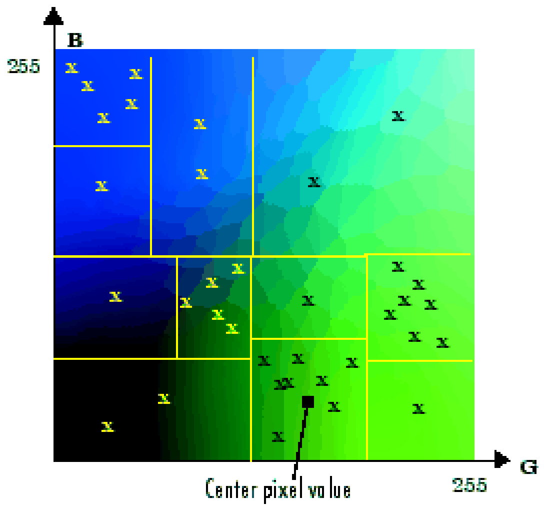
Minimum variance quantization works by associating pixels into groups based on the variance between their pixel values. For example, a set of blue pixels might be grouped together because they have a small variance from the center pixel of the group.

In minimum variance quantization, the boxes that divide the color cube vary in size, and do not necessarily fill the color cube. If some areas of the color cube do not have pixels, there are no boxes in these areas.

While you set the number of boxes, `n`, to be used by `rgb2ind`, the placement is determined by the algorithm as it analyzes the color data in your image. Once the image is divided into `n` optimally located boxes, the pixels within each box are mapped to the pixel value at the center of the box, as in uniform quantization.

The resulting colormap usually has the number of entries you specify. This is because the color cube is divided so that each region contains at least one color that appears in the input image. If the input image uses fewer colors than the number you specify, the output colormap will have fewer than `n` colors, and the output image will contain all of the colors of the input image.

The following figure shows the same two-dimensional slice of the color cube as shown in the preceding figure (demonstrating uniform quantization). Eleven boxes have been created using minimum variance quantization.



Minimum Variance Quantization on a Slice of the RGB Color Cube

For a given number of colors, minimum variance quantization produces better results than uniform quantization, because it takes into account the actual data. Minimum variance quantization allocates more of the colormap entries to colors that appear frequently in the input image. It allocates fewer entries to colors that appear infrequently. As a result, the accuracy of the colors is higher than with uniform quantization. For example, if the input image has many shades of green and few shades of red, there will be more greens than reds in the output colormap. Note that the computation for minimum variance quantization takes longer than that for uniform quantization.

Colormap Mapping

If you specify an actual colormap to use, `rgb2ind` uses *colormap mapping* (instead of quantization) to find the colors in the specified colormap that best match the colors in the RGB image. This method is useful if you need to create images that use a fixed colormap. For example, if you want to display multiple indexed images on an 8-bit display, you can avoid color problems by mapping them all to the same colormap. Colormap mapping produces a good approximation if the specified colormap has similar colors to those in the RGB image. If the colormap does not have similar colors to those in the RGB image, this method produces poor results.

This example illustrates mapping two images to the same colormap. The colormap used for the two images is created on the fly using the MATLAB function `colorcube`, which creates an RGB colormap containing the number of colors that you specify. (`colorcube` always creates the same colormap for a given number of colors.) Because the colormap includes colors all throughout the RGB color cube, the output images can reasonably approximate the input images.

```
RGB1 = imread('autumn.tif');
RGB2 = imread('peppers.png');
X1 = rgb2ind(RGB1,colorcube(128));
X2 = rgb2ind(RGB2,colorcube(128));
```

Note The function `subimage` is also helpful for displaying multiple indexed images. For more information, see or the reference page for `subimage`.

Reducing Colors Using `imapprox`

Use `imapprox` when you need to reduce the number of colors in an indexed image. `imapprox` is based on `rgb2ind` and uses the same approximation methods. Essentially, `imapprox` first calls `ind2rgb` to convert the image to RGB format, and then calls `rgb2ind` to return a new indexed image with fewer colors.

For example, these commands create a version of the `trees` image with 64 colors, rather than the original 128.

```
load trees
[Y,newmap] = imapprox(X,map,64);
imshow(Y, newmap);
```

The quality of the resulting image depends on the approximation method you use, the range of colors in the input image, and whether or not you use dithering. Note that different methods work better for different images. See “Dithering” on page 14-11 for a description of dithering and how to enable or disable it.

Dithering

When you use rgb2ind or imapprox to reduce the number of colors in an image, the resulting image might look inferior to the original, because some of the colors are lost. rgb2ind and imapprox both perform *dithering* to increase the apparent number of colors in the output image. *Dithering* changes the colors of pixels in a neighborhood so that the average color in each neighborhood approximates the original RGB color.

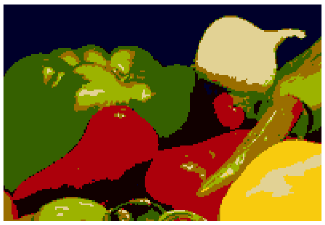
For an example of how dithering works, consider an image that contains a number of dark orange pixels for which there is no exact match in the colormap. To create the appearance of this shade of orange, dithering selects a combination of colors from the colormap, that, taken together as a six-pixel group, approximate the desired shade of orange. From a distance, the pixels appear to be the correct shade, but if you look up close at the image, you can see a blend of other shades. To illustrate dithering, the following example loads a 24-bit truecolor image, and then uses rgb2ind to create an indexed image with just eight colors. The first example does not use dithering, the second does use dithering.

1. Read image and display it.

   ```
   rgb=imread('onion.png');
   imshow(rgb);
   ```

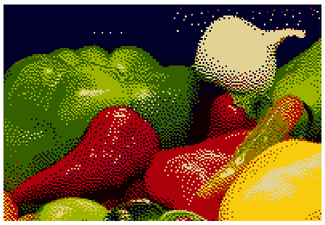
2. Create an indexed image with eight colors and without dithering.

   ```
   [X_no_dither,map]= rgb2ind(rgb,8,'nodither');
   figure, imshow(X_no_dither,map);
   ```



3 Create an indexed image using eight colors with dithering. Notice that the dithered image has a larger number of apparent colors but is somewhat fuzzy-looking. The image produced without dithering has fewer apparent colors, but an improved spatial resolution when compared to the dithered image. One risk in doing color reduction without dithering is that the new image can contain false contours.

```
[X_dither,map]=rgb2ind(rgb,8,'dither');
figure, imshow(X_dither,map);
```



References

[1] J. Campbell, "Speaker recognition: a tutorial," *Proc. IEEE*, vol. 85, pp. 1437–1462, Sept. 1997.

[2] D. A. Reynolds, T. Quatieri, and R. Dunn, "Speaker verification using adapted Gaussian mixture models," *Digital Signal Processing*, vol. 10, no. 1–3, pp. 19–41, 2000.

[3] D. A. Reynolds, "Comparison of background normalization methods for text-independent speaker verification," in *Proc. Eurospeech*, 1997.

[4] D. A. Reynolds and R. C. Rose, "Robust text-independent speaker identification using Gaussian mixture speaker models," *IEEE Trans. Speech Audio Processing*, vol. 3, no. 1, pp. 72–83, 1995.

[5] J. L. Gauvain and C.-H. Lee, "Maximum a posteriori estimation for multivariate Gaussian mixture observations of Markov chains," *IEEE Trans. Speech Audio Processing*, vol. 2, pp. 291–298, Apr. 1994.

[6] E. Bocchieri, "Vector quantization for the efficient computation of continuous density likelihoods," in *Proc. Int. Conf. Acoustics, Speech, Signal Processing*, 1993, pp. 692–695.

[7] K. M. Knill, M. J. F. Gales, and S. J. Young, "Use of Gaussian selection in large vocabulary continuous speech recognition using HMMs," in *Proc. Int. Conf. Spoken Language Processing*, 1996.

[8] D. B. Paul, "An investigation of Gaussian shortlists," in *Proc. Automatic Speech Recognition and Understanding Workshop*, 1999.

[9] T. Watanabe, K. Shinoda, K. Takagi, and K.-I. Iso, "High speed speech recognition using tree-structured probability density function," in *Proc. Int. Conf. Acoustics, Speech, Signal Processing*, 1995.

[10] J. Simonin, L. Delphin-Poulat, and G. Damnati, "Gaussian density tree structure in a multi-Gaussian HMM-based speech recognition system,"

[11] T. J. Hanzen and A. K. Halberstadt, "Using aggregation to improve the performance of mixture Gaussian acoustic models," in *Proc. Int. Conf. Acoustics, Speech, Signal Processing*, 1998.

[12] M. Padmanabhan, L. R. ahl, and D. Nahamoo, "Partitioning the feature space of a classifier with linear hyperplanes," *IEEE Trans. Speech Audio Processing*, vol. 7, no. 3, pp. 282–288, 1999.
in *Proc. Int. Conf. Spoken Language Processing*, 1998.

[13] R. Auckenthaler and J. Mason, "Gaussian selection applied to text-independent speaker verification," in *Proc. A Speaker Odyssey—Speaker Recognition Workshop*, 2001.

[14] J. McLaughlin, D. Reynolds, and T. Gleason, "A study of computation speed-ups of the GMM-UBM speaker recognition system," in *Proc. Eurospeech*, 1999.

[15] S. van Vuuren and H. Hermansky, "On the importance of components of the modulation spectrum of speaker verification," in *Proc. Int. Conf. Spoken Language Processing*, 1998.

[16] B. L. Pellom and J. H. L. Hansen, "An efficient scoring algorithm for Gaussian mixture model based speaker identification," *IEEE Signal Processing Lett.*, vol. 5, no. 11, pp. 281–284, 1998.

[17] J. Oglesby and J. S. Mason, "Optimization of neural models for speaker identification," in *Proc. Int. Conf. Acoustics, Speech, Signal Processing*, 1990, pp. 261–264.

[18] Y. Bengio, R. De Mori, G. Flammia, and R. Kompe, "Global optimization of a neural network—hidden Markov model hybrid," *IEEE Trans. Neural Networks*, vol. 3, no. 2, pp. 252–259, 1992.

[19] H. Bourlard and C. J. Wellekins, "Links between Markov models and multilayer perceptrons," *IEEE Trans. Pattern Anal. Machine Intell.*, vol. 12, pp. 1167–1178, Dec. 1990.

[20] J. Navrátil, U. V. Chaudhari, and G. N. Ramaswamy, "Speaker verification using target and background dependent linear transforms and multi-system fusion," in *Proc. Eurospeech*, 2001.

[21] L. P. Heck, Y. Konig, M. K. Sonmez, and M. Weintraub, "Robustness to telephone handset distortion in speaker recognition by discriminative feature design," *Speech Commun.*, vol. 31, pp. 181–192, 2000.

[22] A. Dempster, N. Laird, and D. Rubin, "Maximum likelihood from incomplete data via the EM algorithm," *J. R. Statist. Soc.*, vol. 39, pp. 1–38, 1977.

[23] K. Shinoda and C. H. Lee, "A structural Bayes approach to speaker adaptation," *IEEE Trans. Speech Audio Processing*, vol. 9, no. 3, pp. 276–287, 2001.

[24] K. Fukunaga, *Introduction to Statistical Pattern Recognition*. New York: Academic, 1990.

[25] J. C. Junqua, *Robust Speech Recogntion in Embedded Systems and PC*

[26] U. V. Chaudhari, J. Navrátil, S. H. Maes, and R. A. Gopinath, "Transformation enhanced multi-grained modeling for text-independent speaker recognition," in *Proc. Int. Conf. Spoken Language Processing*, 2000.

[27] Q. Lin, E.-E. Jan, C. W. Che, D.-S. Yuk, and J. Flanagan, "Selective use of the speech spectrum and a VQGMM method for speaker identification," in *Proc. Int. Conf. Spoken Language Processing*, 1996.

[28] S. Raudys, *Statistical and Neural Classifiers: An Integrated Approach to Design*. New York: Springer, 2001.

[29] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," in *Parallel Distributed Processing*. Cambridge, MA: MIT Press, 1986, pp. 318–364.

[30] [Online] Available: http://www.nist.gov/speech/tests/spk/index.htm.

[31] J. Pelecanos and S. Sridharan, "Feature warping for robust speaker verification," in *Proc. A Speaker Odyssey—Speaker Recognition Workshop*, 2001.

[32] B. Xiang, U. V. Chaudhari, J. Navrátil, N. Ramaswamy, and R. A. Gopinath, "Short-time Gaussianization for robust speaker verification," in *Proc. Int. Conf. Acoustics, Speech, Signal Processing*, 2002.

[33] G. R. Doddington, M. A. Przybocki, A. F. Martin, and D. A. Reynolds, "The NIST speaker recognition evaluation—overview, methodology, systems, results, perspective," *Speech Communication*, vol. 31, pp. 225–254, 2000.

Bing Xiang (M'03) was born in 1973 in China. He received the B.S. degree in radio and electronics and M.E. degree in signal and information processing from Peking University in 1995 and 1998, respectively. In January, 2003, he received the Ph.D. degree in electrical engineering from Cornell University, Ithaca, NY.

From 1995 to 1998, he worked on speaker recognition and auditory modeling in National Laboratory on Machine Perception, Peking University. Then he entered Cornell University and worked on speaker recognition and speech recognition in DISCOVER Lab as a Research Assistant. He also worked in the Human Language Technology Department of IBM Thomas J. Watson Research Center as a summer intern in both 2000 and 2001. He was a selected remote member of the SuperSID Group in the 2002 Johns Hopkins CLSP summer workshop in which he worked on speaker verification with high-lelvel information. In January, 2003, he joined the Speech and Language Processing Department of BBN Technologies where he is presently a Senior Staff Consultant-Technology. His research interests include large vocabulary speech recognition, speaker recognition, speech synthesis, keyword spotting, neural networks and statistical pattern recognition.

Toby Berger (S'60–M'66–SM'74–F'78) was born in New York, NY, on September 4, 1940. He received the B.E. degree in electrical engineering from Yale University, New Haven, CT in 1962, and the M.S. and Ph.D. degrees in applied mathematics from Harvard University, Cambridge, MA in 1964 and 1966, respectively.

From 1962 to 1968 he was a Senior Scientist at Raytheon Company, Wayland, MA, specializing in communication theory, information theory, and coherent signal processing. In 1968 he joined the faculty of Cornell University, Ithaca, NY where he is presently the Irwin and Joan Jacobs Professor of Engineering. His research interests include information theory, random fields, communication networks, wireless communications, video compression, voice and signature compression and verification, neuroinformation theory, quantum information theory, and coherent signal processing. He is the author/co-author of Rate Distortion Theory: A Mathematical Basis for Data Compression, Digital Compression for Multimedia: Principles and Standards, and Information Measures for Discrete Random Fields.

Dr. Berger has served as editor-in-chief of the IEEE Transactions on Information Theory and as president of the IEEE Information Theory