ImageProcessing

This introduction to MATLAB image processing illustrates how to import, display, export, crop, access and transform images, and how to find circles and edges in images. Many words in this Live Script are clickable. Try clicking images.

Author: D. Carlsmith

Image import, export and file information

```
clear; close all;
```

List supported formats with imformats.

```
imformats
```

```
EXT       ISA     INFO        READ      WRITE     ALPHA  DESCRIPTION
------------------------------------------------------------------------------------
bmp       isbmp   imbmpinfo   readbmp   writebmp  0      Windows Bitmap
cur       iscur   imcurinfo   readcur             1      Windows Cursor resources
fts fits  isfits  imfitsinfo  readfits            0      Flexible Image Transport System
gif       isgif   imgifinfo   readgif   writegif  0      Graphics Interchange Format
hdf       ishdf   imhdfinfo   readhdf   writehdf  0      Hierarchical Data Format
ico       isico   imicoinfo   readico             1      Windows Icon resources
j2c j2k   isjp2   imjp2info   readjp2   writej2c  0      JPEG 2000 (raw codestream)
jp2       isjp2   imjp2info   readjp2   writejp2  0      JPEG 2000 (Part 1)
jpf jpx   isjp2   imjp2info   readjp2             0      JPEG 2000 (Part 2)
jpg jpeg  isjpg   imjpginfo   readjpg   writejpg  0      Joint Photographic Experts Group
pbm       ispbm   impnminfo   readpnm   writepnm  0      Portable Bitmap
pcx       ispcx   impcxinfo   readpcx   writepcx  0      Windows Paintbrush
pgm       ispgm   impnminfo   readpnm   writepnm  0      Portable Graymap
png       ispng   impnginfo   readpng   writepng  1      Portable Network Graphics
pnm       ispnm   impnminfo   readpnm   writepnm  0      Portable Any Map
ppm       isppm   impnminfo   readpnm   writepnm  0      Portable Pixmap
ras       isras   imrasinfo   readras   writeras  1      Sun Raster
tif tiff  istif   imtifinfo   readtif   writetif  0      Tagged Image File Format
xwd       isxwd   imxwdinfo   readxwd   writexwd  0      X Window Dump
```

The default (no arguments) `imformats` command lists the names of the MATLAB functions used to read and write each kind of image file. We can view any function using type. This is useful for drilling into MATLAB supplied functions to see exactly what they do.

Let's take a look at a bit map format.

```
type readbmp
```

```
function [X,map] = readbmp(filename)
%READBMP Read image data from a BMP file.
%   [X,MAP] = READBMP(FILENAME) reads image data from a BMP file.
%   X is a uint8 array that is 2-D for 1-bit, 4-bit, and 8-bit
%   image data.  X is M-by-N-by-3 for 16-bit, 24-bit and 32-bit image data.
%   MAP is normally an M-by-3 MATLAB colormap, but it may be empty if the
%   BMP file does not contain a colormap.
%
```

```
%   See also IMREAD, IMWRITE, IMFINFO.

%   Steven L. Eddins, June 1996
%   Copyright 1984-2013 The MathWorks, Inc.

info = imbmpinfo(filename);

map = info.Colormap;
X = readbmpdata(info);
return;
```

The function `readbmp` calls `imbmpinfo` to extract file information and the colormap and then a function that appears to read the actual data. Drill in by type-ing that function.

```
type readbmpdata
```

```
function X = readbmpdata(info)
%READBMPDATA Read bitmap data
%   X = readbmpdata(INFO) reads image data from a BMP file.  INFO is a
%   structure returned by IMBMPINFO. X is a uint8 array that is 2-D for
%   1-bit, 4-bit, and 8-bit image data.  X is M-by-N-by-3 for 24-bit and
%   32-bit image data.

%   Copyright 1984-2006 The MathWorks, Inc.

fid = info.FileID;
offset = info.ImageDataOffset;
width = info.Width;
height = info.Height;


status = fseek(fid,offset,'bof');
if status==-1
  error(message('Spcuilib:scopes:ErrorInvalidDataOffset'));
end

switch info.CompressionType

 case 'none'

  switch info.BitDepth
   case 1
    X = logical(bmpReadData1(fid, width, height));

   case 4
    X = bmpReadData4(fid, width, height);

   case 8
    X = bmpReadData8(fid, width, height);

   case 16
    X = bmpReadData16(fid, width, height);

   case 24
    X = bmpReadData24(fid, width, height);

   case 32
    X = bmpReadData32(fid, width, height);

  end

 case '8-bit RLE'
```

```
  X = bmpReadData8RLE(fid, width, height);

 case '4-bit RLE'
  X = bmpReadData4RLE(fid, width, height);

 case 'bitfields'
  error(message('Spcuilib:scopes:ErrorBitfieldCompressionUnsupported'));

 case 'Huffman 1D'
  error(message('Spcuilib:scopes:ErrorHuffmanCompressionUnsupported'));

 case '24-bit RLE'
  error(message('Spcuilib:scopes:ErrorRLECompressionUnsupported'));

end

%%%
%%% bmpReadData8 --- read 8-bit bitmap data
%%%
function X = bmpReadData8(fid, width, height)

% NOTE: BMP files are stored so that scanlines use a multiple of 4 bytes.
paddedWidth = 4*ceil(width/4);

X = fread(fid,paddedWidth*abs(height),'*uint8');

count = length(X);
if (count ~= paddedWidth*abs(height))
    warning(message('Spcuilib:scopes:WarnInvalidBMP'));
    % Fill in the missing values with zeros.
    X(paddedWidth*abs(height)) = 0;
end

if height>=0
  X = rot90(reshape(X, paddedWidth, height));
else
  X =reshape(X, paddedWidth, abs(height))';
end

if (paddedWidth ~= width)
    X = X(:,1:width);
end



%%%
%%% bmpReadData8RLE --- read 8-bit RLE-compressed bitmap data
%%%
function X = bmpReadData8RLE(fid, width, height)

% NOTE: BMP files are stored so that scanlines use a multiple of 4 bytes.
paddedWidth = 4*ceil(width/4);

inBuffer = fread(fid,'*uint8');

X = bmpdrle(inBuffer, paddedWidth, abs(height), 'rle8');

if height>=0
  X = rot90(X);
else
  X = X';
end
if (paddedWidth ~= width)
    X = X(:,1:width);
end
```

```
%%%
%%% bmpReadData4 --- read 4-bit bitmap data
%%%
function X = bmpReadData4(fid, width, height)

% NOTE: BMP files are stored so that scanlines use a multiple of 4 bytes.
paddedWidth = 8*ceil(width/8);
numBytes = paddedWidth * abs(height) / 2; % evenly divides because of padding

XX = fread(fid,numBytes,'*uint8');

count = length(XX);
if (count ~= numBytes)
    warning(message('Spcuilib:scopes:WarnInvalidBMP'));
    % Fill in the missing values with zeros.
    X(numBytes) = 0;
end
XX = reshape(XX, paddedWidth / 2, abs(height));

X = repmat(uint8(0), paddedWidth, abs(height));
X(1:2:end,:) = bitslice(XX,5,8);
X(2:2:end,:) = bitslice(XX,1,4);

if height>=0
  X = rot90(X);
else
  X = X';
end
if (paddedWidth ~= width)
    X = X(:,1:width);
end


%%%
%%% bmpReadData4RLE --- read 4-bit RLE-compressed bitmap data
%%%
function X = bmpReadData4RLE(fid, width, height)

% NOTE: BMP files are stored so that scanlines use a multiple of 4 bytes.
paddedWidth = 8*ceil(width/8);
%numBytes = paddedWidth * abs(height) / 2; % evenly divides because of padding

inBuffer = fread(fid,'*uint8');

if height>=0
  X = rot90(bmpdrle(inBuffer, paddedWidth, abs(height), 'rle4'));
else
  X = bmpdrle(inBuffer, paddedWidth, abs(height), 'rle4')';
end
if (paddedWidth ~= width)
    X = X(:,1:width);
end


%%%
%%% bmpReadData1 --- read 1-bit bitmap data
%%%
function X = bmpReadData1(fid, width, height)

% NOTE: BMP files are stored so that scanlines use a multiple of 4 bytes.
paddedWidth = 32*ceil(width/32);
numPixels = paddedWidth * abs(height);  % evenly divides because of padding
```

```
[X, count] = fread(fid,paddedWidth*abs(height),'*ubit1');

if (count ~= numPixels)

    % Fill in the missing values with zeros.
    X(numPixels) = 0;
end
X = reshape(X, paddedWidth, abs(height));

if height>=0
  X = rot90(X);
else
  X = X';
end

if (paddedWidth ~= width)
    X = X(:,1:width);
end

%%%
%%% bmpReadData16 --- read 16-bit grayscale data
%%%
function X = bmpReadData16(fid, width, height)

% NOTE: BMP files are stored so that scanlines use a multiple of 4 bytes.
paddedWidth = 4*ceil(width/4);

X = fread(fid,paddedWidth*abs(height),'*uint16');

count = length(X);
if (count ~= paddedWidth*abs(height))
    warning(message('Spcuilib:scopes:WarnInvalidBMP'));
    % Fill in the missing values with zeros.
    X(paddedWidth*abs(height)) = 0;
end

if height>=0
  X = rot90(reshape(X, paddedWidth, height));
else
  X = reshape(X, paddedWidth, abs(height))';
end

if (paddedWidth ~= width)
    X = X(:,1:width);
end


%%%
%%% bmpReadData24 --- read 24-bit bitmap data
%%%
function RGB = bmpReadData24(fid, width, height)

% NOTE: BMP files are stored so that scanlines use a multiple of 4 bytes.
byteWidth = 3*width;
paddedByteWidth = 4*ceil(byteWidth/4);
numBytes = paddedByteWidth * abs(height);

X = fread(fid,numBytes,'*uint8');

count = length(X);
if (count ~= numBytes)
    warning(message('Spcuilib:scopes:WarnInvalidBMP'));
    % Fill in the missing values with zeros.
    X(numBytes) = 0;
```

```
end

if height>=0
  X = rot90(reshape(X, paddedByteWidth, abs(height)));
else
  X = reshape(X, paddedByteWidth, abs(height))';
end

if (paddedByteWidth ~= byteWidth)
    X = X(:,1:byteWidth);
end

RGB(1:abs(height), 1:width, 3) = X(:,1:3:end);
RGB(:, :, 2) = X(:,2:3:end);
RGB(:, :, 1) = X(:,3:3:end);


%%%
%%% bmpReadData32 --- read 32-bit bitmap data
%%%
function RGB = bmpReadData32(fid, width, height)

% NOTE: BMP files are stored so that scanlines use a multiple of 4 bytes.
byteWidth = 4*width;
paddedByteWidth = 4*ceil(byteWidth/4);
numBytes = paddedByteWidth * abs(height);

X = fread(fid,numBytes,'*uint8');

count = length(X);
if (count ~= numBytes)
    warning(message('Spcuilib:scopes:WarnInvalidBMP'));
    % Fill in the missing values with zeros.
    X(numBytes) = 0;
end

if height>=0
  X = rot90(reshape(X, paddedByteWidth, abs(height)));
else
  X = reshape(X, paddedByteWidth, abs(height))';
end

if (paddedByteWidth ~= byteWidth)
    X = X(:,1:byteWidth);
end

RGB(1:abs(height), 1:width, 3) = X(:,1:4:end);
RGB(:, :, 2) = X(:,2:4:end);
RGB(:, :, 1) = X(:,3:4:end);
```

We are down in the weeds. Any computer file is essentially a list of (possibly not physicially contiguous) words, the size of which is defined by the physical architecture of the computer. A 64-bit word for example contains eight 8-bit bytes. A pixel value, say some red pixel value created generated by the sensor readout chip, is conventionally subject to so-called gamma correction, a nonlinear transformation to an 8-bit value. Eight of these can fit into one 64-bit word. Different formats pack the pixel values in different ways into the bytes in the words of a file. One requires different codes to unpack the different formats in any general graphics application. MATLAB core libraries provide such functions for common formats. Thanks MATLAB. If you have to work with some uncommon format used in some science application, for example data from a huge array of sensors at the focal point of a new telescope in space, you will have to hunt up or write your own unpacking algorithm, or find code to convert to some standard format such as FITS used in the industry.

Open an image file from the internet using imread. We first create a character vector to hold the filename. If we use a URL, imread will recognize that. Else imread will attempt to find a file with that name in your local path.

Try this: Use your own URL to play with a different image.

```
fln= 'https://live.staticflickr.com/5055/5447218318_a1ce473203_o_d.jpg';
RGB=imread(fln);
```

Try this: If you hover you cursor over a variable, MATLAB will display information about the variable. Hover over the variable `fln` above.

MATLAB `imread` used the file extension to invoke the appropriate image unpacking functions and filled and returned an array which we caught and named RGB. Thanks MATLAB!

Display the image with imshow.

Tip: This next line is commented out to suppress inclusion of a large file in Live Script which can slow it down. If Live Script slows down, try execting it from the command line not the Live Script Editor. If executed from the comand line, you can terminate execution with a control_C. Also, you can clear all Live Script output under the view menu to clean up a situation.

Try this: Try the next command at the command line.

```
% imshow(RGB)
```

Show information included with the image file using imfinfo.

```
flninfo=imfinfo(fln)
```

```
flninfo = struct with fields:
            Filename: 'https://live.staticflickr.com/5055/5447218318_a1ce473203_o_d.jpg'
         FileModDate: '20-Aug-2019 16:10:02'
            FileSize: 3249307
              Format: 'jpg'
       FormatVersion: ''
               Width: 4288
              Height: 2848
            BitDepth: 24
           ColorType: 'truecolor'
     FormatSignature: ''
     NumberOfSamples: 3
        CodingMethod: 'Huffman'
      CodingProcess: 'Sequential'
             Comment: {}
                Make: 'NIKON CORPORATION'
               Model: 'NIKON D5000'
         Orientation: 1
         XResolution: 300
         YResolution: 300
      ResolutionUnit: 'Inch'
            Software: 'Adobe Photoshop Elements 7.0 Windows'
            DateTime: '2011:02:14 20:37:31'
    YCbCrPositioning: 'Co-sited'
```

```
       DigitalCamera: [1×1 struct]
             GPSInfo: [1×1 struct]
       ExifThumbnail: [1×1 struct]
```

Most image files include provenance information using standards like EXIF. We discovered this image, lifted from the internet, was taken with a Nikon D5000 and processed with Photoshop. The function `imfinfo` returned a MATLAB structure array that itself contains structure arrays for DigitalCamera and GPS information and a Thumbnail (tiny preview).

Try this: You can drill into a structure by double clicking its name in the workspace pane.

We can address elements of a structure using a dot notation structureName.substructureName.

Get digital camera info.

```
flninfo.DigitalCamera
```

```
ans = struct with fields:
               ExposureTime: 0.016667
                    FNumber: 4
              ExposureProgram: 'Not defined'
              ISOSpeedRatings: 400
                  ExifVersion: [48 50 50 49]
            DateTimeOriginal: '2010:11:02 18:28:31'
           DateTimeDigitized: '2010:11:02 18:28:31'
      ComponentsConfiguration: 'YCbCr'
       CompressedBitsPerPixel: 4
            ExposureBiasValue: 0
             MaxApertureValue: 3.6
                 MeteringMode: 'Pattern'
                  LightSource: 'unknown'
                        Flash: 'Flash did not fire, no strobe return detection function, auto flash mode, flash funct
                  FocalLength: 18
                  UserComment: [1×44 double]
                   SubsecTime: '00'
          SubsecTimeOriginal: '00'
         SubsecTimeDigitized: '00'
             FlashpixVersion: [48 49 48 48]
                  ColorSpace: 'sRGB'
            CPixelXDimension: 4288
            CPixelYDimension: 2848
         InteroperabilityIFD: [1×1 struct]
               SensingMethod: 'One-chip color area sensor'
                  FileSource: 'DSC'
                   SceneType: 'A directly photographed image'
                  CFAPattern: [0 2 0 2 1 2 0 1]
              CustomRendered: 'Normal process'
                ExposureMode: 'Auto exposure'
                WhiteBalance: 'Auto white balance'
            DigitalZoomRatio: 1
        FocalLengthIn35mmFilm: 27
             SceneCaptureType: 'Standard'
                 GainControl: 'Low gain up'
                    Contrast: 'Normal'
                  Saturation: 'Normal'
                   Sharpness: 'Normal'
        SubjectDistanceRange: 'unknown'
```

Show workspace array information with whos.

```
whos RGB
```

```
  Name         Size                    Bytes  Class    Attributes

  RGB       2848x4288x3             36636672  uint8
```

Write image to file with imwrite.

```
imwrite(RGB,'balls.jpg','jpg')
```

Get image array dimensions using size.

```
RGBsize=size(RGB)
```

```
RGBsize = 1×3
        2848        4288           3
```

Crop an image

Create indices corresponding to horizontal and vertical middle fraction of image.

Try this: Customize the horizontal column (h1, h2) and vertical row (v1,v2) ranges to create your own cropped image.

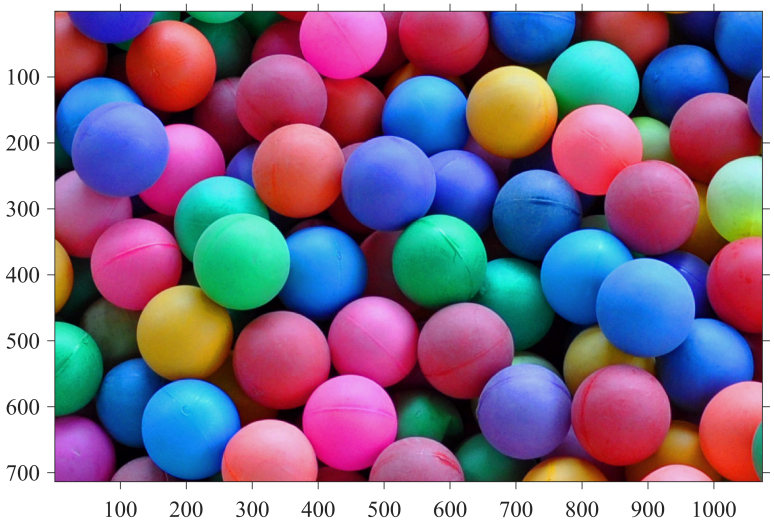
```
n=4;h1=int16(RGBsize(1)*(1/2-1/(2*n)));h2=int16(RGBsize(1)*(1/2+1/(2*n)));
v1=int16(RGBsize(2)*(1/2-1/(2*n)));v2=int16(RGBsize(2)*(1/2+1/(2*n)));
```

Create smaller cropped image, sufficient to illustrate image processing, simply by addressing ranges of pixels using the colon operator.

```
RGBcropped=RGB(h1:h2,v1:v2,:);
```

Display cropped image.

```
imshow(RGBcropped)
```

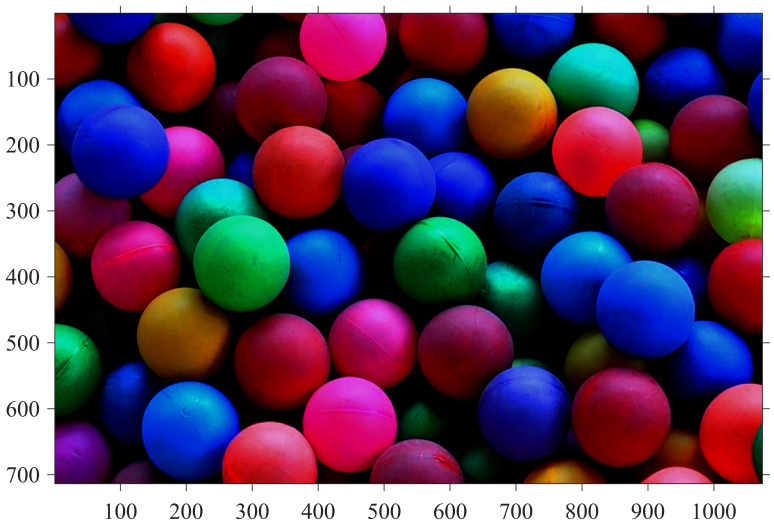


Access and transform color

Adjust the contrast of the cropped image, specifying contrast limits to imadjust.

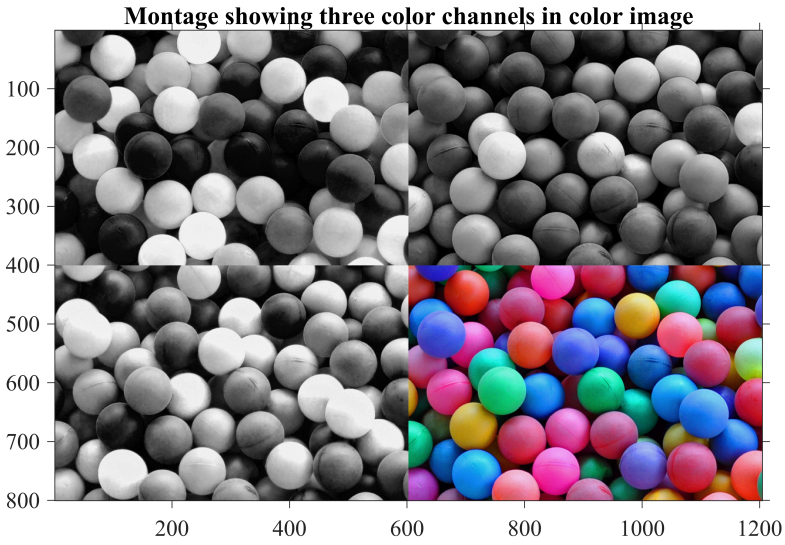
This operation is a nonlinear scaling of values for each channel using a conventional gamma correction of the form $V_{out} = AV_{in}^{\gamma}$ developed to compress image sensor values into 8 bits. Different sensors have different relative sensitivites in sampling light as RGB values. To approximately reproduce color correctly, different display technologies need to convert digital image values to yet different technology-dependent RGB values to drive light emitting elements. Several standards are recognized to facilitate the optimal transfer of information.

```
gamma=[2.,2.,2.];
RGB2 = imadjust(RGBcropped,[.2 .2 .2; 1 1 1],[],gamma);
figure
imshow(RGB2)
```



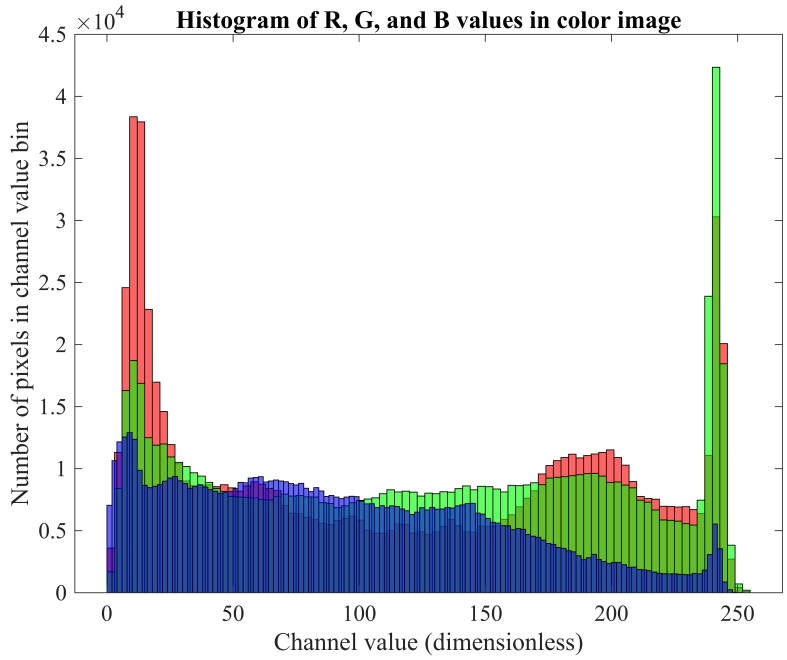
Show montage of three different color channels.

```
R=RGBcropped(:,:,1);G=RGBcropped(:,:,2);B=RGBcropped(:,:,3);
montage({R,G,B,RGBcropped})
title('Montage showing three color channels in color image')
```



Histogram the values of the R, G, and B channels. We choose a different display color for each channel. You may notice the display color for R might not look exactly 'red' to you on your display. Maybe your display colors are not correctly set.

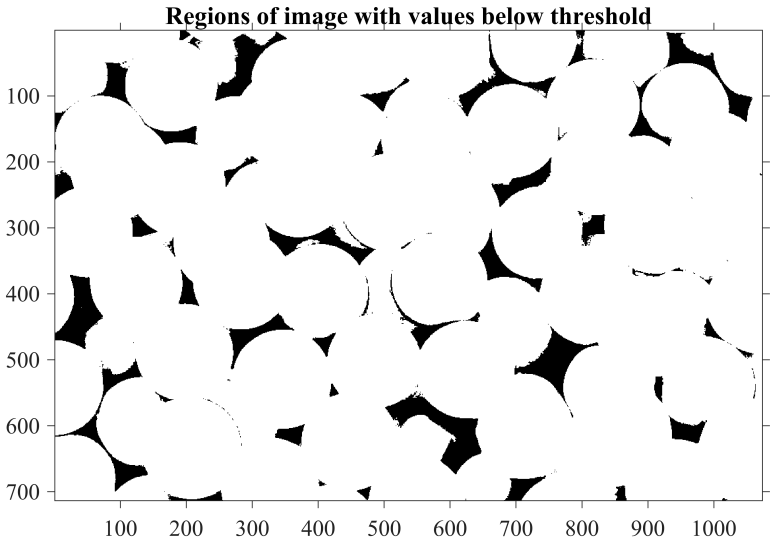
```
histogram(R,'facecolor',[1 0 0]); hold on;
histogram(B,'facecolor',[0 1 0]);
histogram(G,'facecolor',[0 0 1]);
title('Histogram of R, G, and B values in color image')
xlabel('Channel value (dimensionless)')
ylabel('Number of pixels in channel value bin')
hold off
```



Accessing an image by value

Show the parts of the image close to black by creating a mask to select pixels with values less than a threshold. The inequality RGBcropped(:,:,1)<threshold returns an array of logical values true for array elements for which the inequality is valid, false otherwise. The symbol & represents a logical AND operation. See MATLAB Logical Operations. The logical array darkmask is converted to a binary array (1's and 0's) for display as an image.

```
threshold=50;
darkmask=RGBcropped(:,:,1)<threshold&RGBcropped(:,:,2)<threshold...
    &RGBcropped(:,:,3)<threshold;
dark=ones(size(RGBcropped(:,:,1) ));
dark(darkmask)=0;
figure
imshow(dark)
title('Regions of image with values below threshold')
```



Color space transformations

Convert the RGB image to the HSV colorspace by using the rgb2hsv function. H (hue) starts at pure R and increases with R dominant towards G then with G dominant towards B and ends at pure B. Assume RGB are normalized to 1. S (saturation) is S=1-MIN(RGB)/MAX(RGB) so R=G=1/2, B=0 has maximum S. V(value)=MAX(RGB) of the dominate RGB color. The transformation does not distinguish between raw RGB and intensity compressed (gamma-corrected) RGB values.

See https://en.wikipedia.org/wiki/HSL_and_HSV .

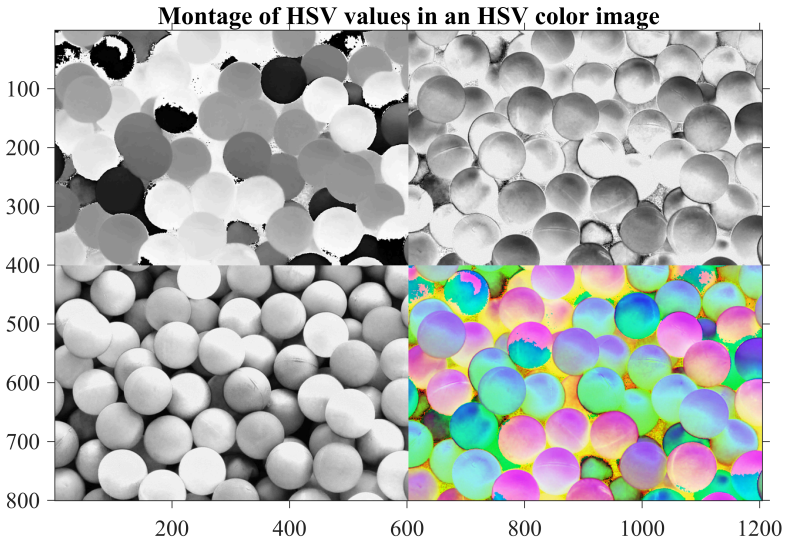
```
hsvImage = rgb2hsv(RGBcropped);
```

Split the HSV image into its component hue, saturation, and value channels with imsplit.

```
[h,s,v] = imsplit(hsvImage);
```

Display the separate H, S, and V channels as a montage.

```
montage({h,s,v,hsvImage})
title('Montage of HSV values in an HSV color image')
```



Segment image using `findcircles`.

The MATLAB function imfindcircles requires values for several search parameters including a range of radii. In using this function on a particular image, you will need to choose appropriate values so the function neither converts every speck of pixels into a circle nor searches only for circles larger than any in the image.

The following invocation of `imfindcircles` uses several arguments. The first is the image array itself, the secnd a range of radii, the rest invoked by name. To understand a function call like this, you must study the documentation for the function. Search for it using help or use and internet search.

MATLAB supports functions with a variable number of arguments and a variable number of returned quantities. The documentation of each function shows you the various ways the function can be used, starting with some minimal form. We will use a certain choice of parameters and catch the locations of the centers of the circles found, the corresponding radii, and corresponding values describing the quality of each found circle. The balls in theimage are all nearly at the same distance so we can use a narrow range of radii.

Try this: Vary the parameter values and methods used in circle finding to see if you can find more circles. In naive use, circle finding proceeds by converting the image to grayscale (total intensity), ignoring color. Different algorithms might first segment by color, but looking closely you will see that light reflected from one ball onto its neighbors make that approach less than straight forward.

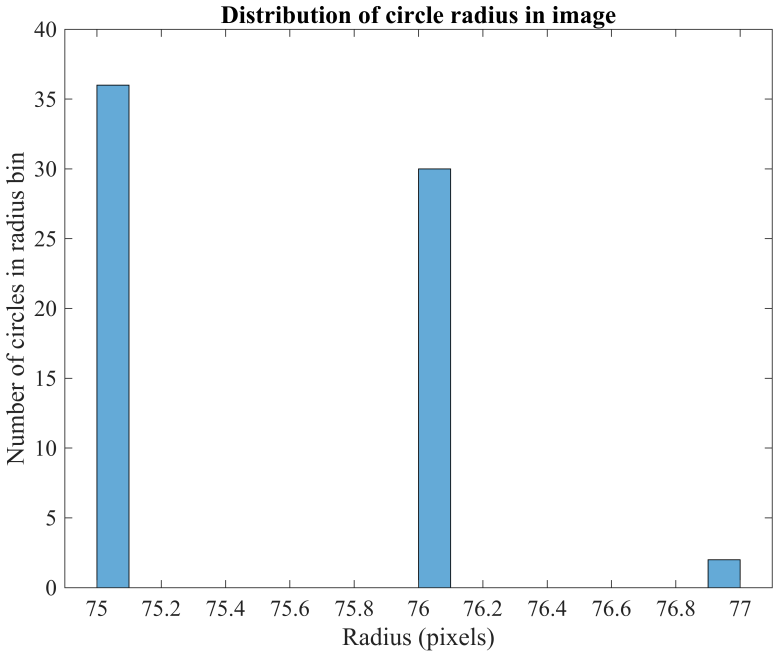
Try this: Switch to segment the HSV not RGB image.

```
X=RGBcropped;
%X=hsvImage;
[centers,radii,metric] = imfindcircles(X,[75 77],'ObjectPolarity',...
    'bright','Sensitivity',1,'EdgeThreshold',0.4,'Method','TwoStage');
```

Histogram the radii of the circles found.

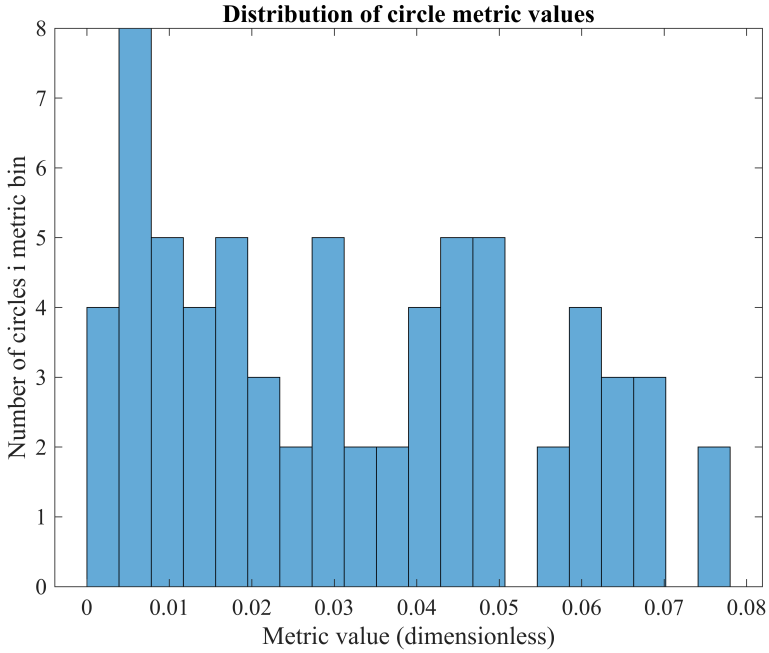
```
nbins=20;
```

```
histogram(radii, nbins)
title('Distribution of circle radius in image')
xlabel('Radius (pixels)'); ylabel('Number of circles in radius bin')
```



Histogram parameter describing circle quality.

```
histogram(metric,nbins);
title('Distribution of circle metric values')
xlabel('Metric value (dimensionless)')
ylabel('Number of circles i metric bin')
```



Circles are returned in decreasing order of quality and value of quality metric. Select circles with large metric values using a logical mask.

```
range=1:length(radii(metric>0.03));
```

The preceding nested functions construct is typical MATLAB. To understand it, start with the innermost function call `metric>0.03.` Firstly, `<` is an elementary function called an operator function that, like + and `-`, has always two arguments/inputs. Rather than write `c=Plus(a,b)`, we can write `c=a +b` with + understood to operate on its immediate left and immediate right arguments. Similarly, `<` is an elementary function. `Now` try

```
>>test=metric>0.03
```

at the command line, a function call equivalent to `test = gt(metric,0.03)` which hasthe form `output=function(input arguments)`. The returned output that we have caught as `test` is a logical array of the same dimesion as `metric` and its elements are logical true or false upon if the corresponding elements of the array `metric` satisfies the condition. The vector `radii` has the same size as `metric.` An expression of the form radii`(select)` when select is a collection of indices returns a reduced size array containing the elements of `radii` with the indices specified in `select`. For example `radii(1:5)` returns the first 5 elements. When `select` is a logical array, radii`(select)` returns the elements of `radii` corresponding to those indices for which `select` is logical true. So `selectradii=radii(metric>0.03)` is a select array of radii satisfying the selection criterion. Now length(selectradii) is the length of that select array and

```
range=1:length(radii(metric>0.03));
```

is a vector of sequential indices from 1 to the number of circles with metric>0.03.

Now `imfindcircles` returns center locations, radii, and metric sorted by metric in decreasing order so the following statements produce reduced lists of these quantities containing values for which the metric is greater than 0.03.

```
centersStrong = centers(range,:);
radiiStrong = radii(range);
metricStrong = metric(range);
```

Here the colon stands for all values of the 2nd index in `centers`.

There is simpler way to do this without relying on sorting by `imfindcircles`, namely to address the arrays using the logical condition array itself. See Find Array Elements that Meet a Condition.

```
mask= metric>0.03;
radiiStrong2=radii(mask);
```

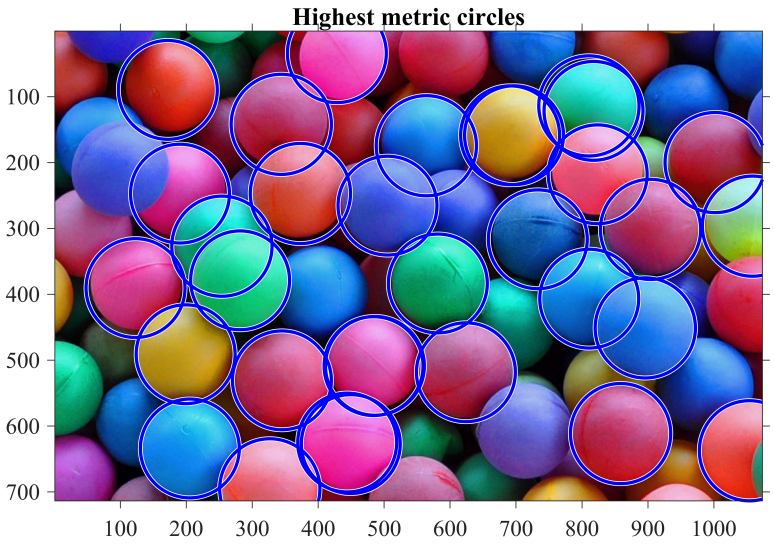
We can test for equality of these arrays using isequal.

```
isequal(radiiStrong, radiiStrong2)
```

```
ans = logical
   1
```

Superpose the highest metric circle perimeters atop the original image. One observes the metric favors nonoccluded balls.

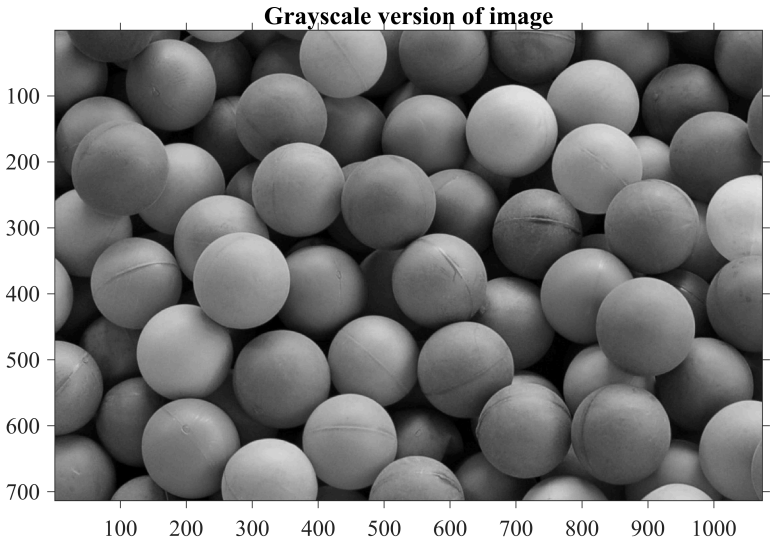
```
imshow(X);
viscircles(centersStrong, radiiStrong,'EdgeColor','b');
title('Highest metric circles')
```



Find edges in an image.

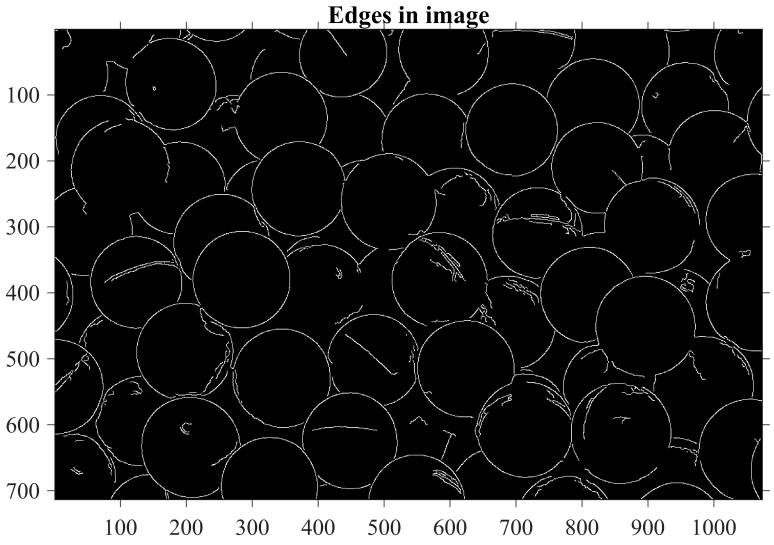
Convert RGBcropped to gray scale (single channel) using rgb2gray.

```
BW=rgb2gray(RGBcropped);
imshow(BW)
title('Grayscale version of image')
```



Find edges in the gray scale image with the Canny algorithm which uses a derivative of a Gaussian filter to find edges with two thresholds. See edge. Other algorithms are available.

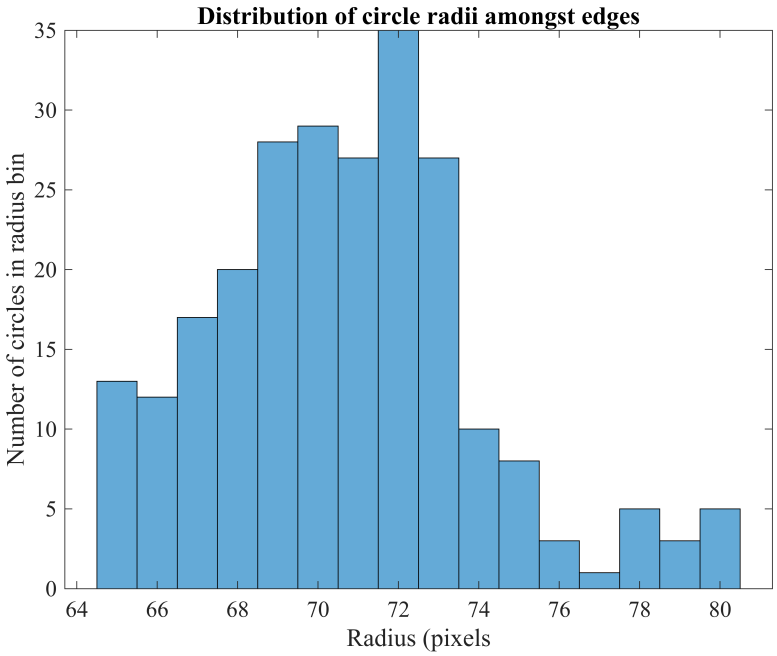
```
[imgCanny,threshOut] = edge(BW,'Canny',0.12019,1.23);
imshow(imgCanny)
title('Edges in image')
```



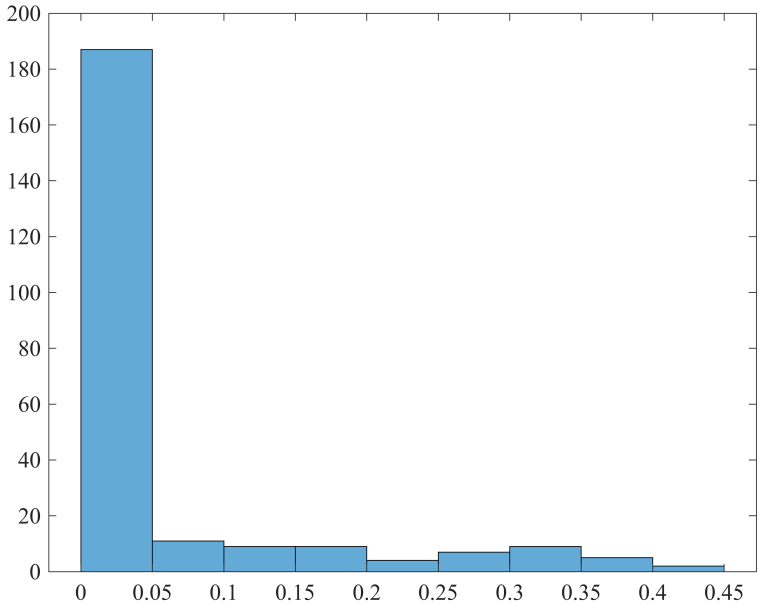
Find circles amongst the edges.

```
[centers, radii, metric] = imfindcircles(imgCanny, [65 80], ...
```

```
    'Sensitivity', 0.9870, ...
    'EdgeThreshold', 0.09, ...
    'Method', 'TwoStage', ...
    'ObjectPolarity', 'Bright');
histogram(radii)
title('Distribution of circle radii amongst edges')
xlabel('Radius (pixels')
ylabel('Number of circles in radius bin')
```




```
histogram(metric)
```



Visualize the strongest circles found in the canny edge data.

```
imshow(imgCanny);
range=1:length(radii(metric>0.03));
centersStrong = centers(range,:);
radiiStrong = radii(range);
metricStrong= metric(range);
viscircles(centersStrong, radiiStrong,'EdgeColor','b');
title('Circles found amongst edges in image')
```