* cv::Mat
* // Do the smoothing
* // ( Note: Could use GaussianBlur(), blur(), medianBlur() or bilateralFilter(). )
* pyrDown
* 2-7 = Canny
* 2-9
  + Vec3b = 3-tuple of RGB
* 2-11 log\_polar coming in ch 11
* 3
  + SparseMat next chapter
  + Vec{2,3,4,6}{b,w,s,i,f,d}
  + Matx{1-6}{1-6}{f,d} for transformations
  + Point{2,3}{i,f,d}
  + Points
    - Dot product, cross product, inside rectangle
  + Size
  + Rect has a point (upper left) and size, but doesn’t inherit from either
    - Intersection
    - Smallest container
    - Translation
    - Enlarge
    - Equality compare
  + Complex numbers, re and im
  + Range class, start -> end integers
  + InputArray OutputArray, the former is assumed to be read only
  + alignPtr() what’s an “aligned pointer”? something to do with block size
* 4
  + It is critical to understand that the data in an array is not attached rigidly to the array object. The cv::Mat object is really a header for a data area, which—in principle—is an entirely separate thing. For example, it is possible to assign one matrix n to another matrix m (i.e., m=n). In this case, the data pointer inside of m will be changed to point to the same data as n
  + Mat constructors
    - Metadata
    - Previous mat
    - Use DataType in some cases to wrap object types
  + Get pointer to specific row w/ ptr<>()
* This is worth reiterating. The purpose of using the template forms cv::Mat\_<> and cv::SparseMat\_<> are so you don’t have to use the template forms of their member functions. Consider this example, where we have a matrix defined by:
* cv::Mat m( 10, 10, CV\_32FC2 );
* Individual element accesses to this matrix would need to specify the type of the matrix, as in the following:
* m.at< Vec2f >( i0, i1 ) = cv::Vec2f( x, y );
* Alternatively, if you had defined the matrix m using the template class, you could use at() without specialization, or even just use operator():
* cv::Mat\_<Vec2f> m( 10, 10 );
* m.at( i0, i1 ) = cv::Vec2f( x, y );
* // or...
* m( i0, i1 ) = cv::Vec2f( x, y );
* There is a great deal of simplification in your code that results from using these template definitions.
* Why Sobel most widely used?
* Sobel derivative in x vs y dimension?
* Laplace edges
* Scharr constant passed to Sobel()
* Discrete fourier transform = frequency representation of original image
* Hough line transform = list of components
* Speed up convolutions with DFT
* Canny edge detection – converts edge-candidate pixels into contours
* The cv::Canny() function expects an input image, which must be single-channel, and an output image, which will also be grayscale (but which will actually be a Boolean image).
* The final argument L2gradient is used to select between computing the directional gradient “correctly” using the proper L2-norm, or using a faster, less accurate L1-norm-based method
* Hough line transform – finds lines in the picture, kind of odd results
* The topic of image segmentation is a large one, which we have touched on in several places already, and will return to in more sophisticated contexts later in the book. Here, we will focus on several methods of the library that specifically implement techniques that are either segmentation methods in themselves, or primitives that will be used later by more sophisticated tactics. Note that, at this time, there is no general “magic” solution for image segmentation, and it remains a very active area in computer vision research. Despite this, many good techniques have been developed that are reliable at least in some specific domain, and in practice can yield very good results.Flood FillFlood fill [Heckbert90; Shaw04; Vandevenne04] is an extremely useful function that is often used to mark or isolate portions of an image for further processing or analysis. Flood fill can also be used to derive, from an input image, masks that can be used by subsequent routines to speed or restrict processing to only those pixels indicated by the mask. The function cv::floodFill() itself takes an optional mask that can be further used to control where filling is done (e.g., for multiple fills of the same image).
* In many practical contexts, we would like to segment an image but do not have the benefit of any kind of separate background mask. One technique that is often effective in this context is the watershed algorithm [Meyer92], which converts lines in an image into “mountains” and uniform regions into “valleys” that can be used to help segment objects. The watershed algorithm first takes the gradient of the intensity image; this has the effect of forming valleys or basins (the low points) where there is no texture, and of forming mountains or ranges (high ridges corresponding to edges) where there are dominant lines in the image. It then successively floods basins starting from caller-specified points until these regions meet. Regions that merge across the marks so generated are segmented as belonging together as the image “fills up.” In this way, the basins connected to the marker point become “owned” by that marker. We then segment the image into the corresponding marked regions.More specifically, the watershed algorithm allows a user (or another algorithm) to mark parts of an object or background that are known to be part of the object or background. Alternatively, the caller can draw a simple line or collection of lines that effectively tells the watershed algorithm to “group points like these together.” The watershed algorithm then segments the image by allowing marked regions to “own” the edge-defined valleys in the gradient image that are connected with the segments
* The Grabcuts algorithm is capable of obtaining excellent segmentations, often with no more than a bounding rectangle around the foreground object to be segmented.The original Graphcuts algorithm used user-labeled foreground and user-labeled background regions to establish distribution histograms for those two classes of image regions. It then combined the assertion that unlabeled foreground or background should conform to similar distributions with the idea that these regions tend to be smooth and connected (i.e., a bunch of blobs). These assertions were then combined into an energy functional that gave a low energy (i.e., cost) to solutions that conformed to these assertions and a high energy to solutions that violated them. The algorithm obtained the final result by minimizing this energy function.22The Grabcuts algorithm extends Graphcuts in several important ways. The first is that it replaces the histogram models with a different (Gaussian mixture) model, enabling the algorithm to work on color images. In addition, it solves the energy functional minimization problem in an iterative manner, which provides better results overall, and allows much greater flexibility in the labeling provided by the user. Notably, this latter point makes possible even one-sided labelings, which identify either only background or only foreground pixels (where Graphcuts required both).
* Given a set of multidimensional data points whose dimensions are (x, y, blue, green, red), mean-shift can find the highest density “clumps” of data in this space by scanning a window over the space. Notice, however, that the spatial variables (x, y) can have very different ranges from the color-magnitude ranges (blue, green, red). Therefore, mean-shift needs to allow for different window radii in different dimensions.
* The output of the mean-shift segmentation algorithm is a new image that has been “posterized,” meaning that the fine texture is removed and the gradients in color are mostly flattened. You can then further segment such images using whatever algorithm is appropriate for your needs (e.g., cv::Canny() combined with cv::findContours(), if in fact a contour segmentation is what you ultimately want
* Histograms can be used to represent such diverse things as the color distribution of an object, an edge gradient template of an object [Freeman95], or the distribution of probabilities representing our current hypothesis about an object’s location
* Histograms that represent continuous distributions do so by quantizing the points into each grid cell.1 This is where problems can arise, as shown in Figure 13-3. If the grid is too wide (upper left), then the output is too coarse and we lose the structure of the distribution. If the grid is too narrow (upper right), then there is not enough averaging to represent the distribution accurately and we get small, “spiky” cells.
* cv::calcHist()
* While histSize indicates the number of bins in each dimension, ranges indicates the values that correspond to each bin in each dimension. ranges also can be a C-style array or an STL vector
* cv::findContours()
* The function cv::findContours() computes contours from binary images. It can take images created by cv::Canny(), which have edge pixels in them, or images created by functions like cv::threshold() or cv::adaptiveThreshold(), in which the edges are implicit as boundaries between positive and negative regions.[1](https://www.safaribooksonline.com/library/view/learning-opencv-3/9781491937983/ch14.html#ch14fn1)
* Polygon approximation with cv::approxPolyDP()
  + Can’t say “find a square”
  + Book says there’s two methods, but not seeing the other
* Video 22 Frequency equalization can increase contrast but also noise
* Lookuptable “lut” precalculate all possible values for a pixel and lookup, rather than running a formula every pixel
  + Cv::LUT
* Video 23 Cartoonize
  + Medianblur to remove noise
  + Canny to detect edges
  + Dilate to connect edges?
  + Blur to smooth edges
  + Bilateral filter to reduce color complexity
* Video 24 object detection
  + Noise removal
    - medianblur
  + Lighting removal
    - Threshold? Go from grayscale to b&w binary
    - How did he remove the background?
      * Separate picture under same lighting conditions, boo
      * Or use multiple images with different objects
      * Or do a large blur to wash out the objects you are trying to find and “create” the background
    - Difference/division methods to remove the light pattern
  + Binarization
    - Threshold above