# Contents

## 1 Histograms and matching
- 1.1 Creating histograms ........................................ 3
- 1.2 Basic manipulations ........................................ 4
- 1.3 Comparing 2 histograms .................................... 4
- 1.4 Example: Calculate Histograms ............................ 5
- 1.5 Earth mover’s distance (EMD) ............................... 7
- 1.6 Back projection ............................................. 8

## 2 Template matching ............................................ 9
- 2.1 Example: template matching ................................. 10

## 3 Contours .................................................................. 15
- 3.1 Memory storage ............................................. 15
- 3.2 Sequences .................................................. 15
- 3.3 Contour Finding ............................................ 16
- 3.4 Drawing Contours .......................................... 17
- 3.5 Contour example ............................................ 17
- 3.6 Summary characteristics ................................... 19
- 3.7 Matching Contours .......................................... 20
- 3.8 Contour convexity and convexity defects ............... 21

## 4 Image parts and segmentation .................................. 23
- 4.1 Background subtraction .................................... 23
- 4.2 Scene modelling ........................................... 23
- 4.3 Comparing Background Methods ......................... 26
- 4.4 The Watershed algorithm .................................. 26
- 4.5 Image repair by Inpainting ................................. 27
- 4.6 Mean-Shift segmentation .................................. 27
- 4.7 Delaunay Triangulation, Voronoi Tesselation .......... 27

## 5 Tracking and motion ............................................... 29
- 5.1 Corner finding .............................................. 29
- 5.2 Subpixel corners ............................................ 30
- 5.3 Invariant features .......................................... 31
- 5.4 Optical Flow ................................................ 31
- 5.5 Dense Tracking Techniques ............................... 31
- 5.6 The Kalman Filter .......................................... 32
5.7 The Condensation Algorithm .................................................. 34

6 Kalman filters ........................................................................... 35
   6.1 The Discrete Kalman Filter .................................................. 35

7 Indices and tables ..................................................................... 37
Histograms can be used to represent such diverse things as the color distribution of an object, the edge gradient template of an object, and the distribution of probabilities representing our current hypothesis about an object location.

**Histograms usage in Computer vision:**

- Scene transitions in video by marking when the edge and color statistics markedly change from frame to frame.
- Identifying interest points in images by assigning each interest point a “tag” consisting of histograms of nearby features.
- Histograms of edges, colors, corners, and so on form a general feature type that is passed to classifiers for object recognition.
- Sequences of color or edge histograms are used to identify whether videos have been copied on the web.
- etc.

Histograms are one of the classic tools of computer vision. They are simply collected counts of the underlying data organized into a set of predefined bins.

Histograms that represent continuous distributions do so by implicitly averaging the number of points in each grid cell. If the grid is too wide, then there is too much averaging and we lose the structure of distribution. If the grid is too narrow, then there is not enough averaging to represent the distribution accurately and we get small “spiky” cells.

The histogram data structure in OpenCV is capable of representing histograms in one or many dimensions, and it contains all data necessary to track bins of both uniform and non-uniform sizes.

### Creating histograms

```python
>>> cv.CreateHist(dims = (INT, INT), type = cv.CV_HIST_ARRAY|cv.CV_HIST_SPARSE [, ranges, uniform = True])
```

- `dims` - how many dimensions the histogram have.
• type - CV_HIST_ARRAY, which is used for multidimensional histograms to be stored using the dense multidimensional matrix structure (CvMatND).

  – CV_HIST_SPARSE, which is used for present data as sparse matrix.

• ranges - for uniform histograms, it is an array of floating-point value pairs, where the number of value pairs is equal to the number of dimensions.

  – for non-uniform histogram, the pairs used by the uniform histogram are replaced by arrays containing the values by which the non-uniform bins are separated.

  – it is possible to set ranges to None, in which case the ranges are simply “unknown”, they can be set later using specialized function `SetHistBinRanges(hist, ranges, uniform = True|False)`

```python
>>> cv.ClearHist(hist) - resets all of the bins to 0
```

### Basic manipulations

- Get floating point number for the value in the appropriate bin:

```python
>>> cv.QueryHistValue_1D(hist, idx0), cv.QueryHistValue_2D(..), cv.QueryHistValue_3D(..)
>>> cv.QueryHistValue_nD(hist, ids = [])
```

- Normalize histogram, so that individual bins will then represent the fraction of the total number of events assigned to the entire histogram:

```python
>>> cv.NormalizeHist(hist, factor)
```

- Append threshold:

```python
>>> cv.ThresHist(hist, cutoff_factor)
```

- Get minimal and maximal values found in the histogram:

```python
>>> cv.GetMinMaxHistValue(hist, min_val = double, max_val = double, [min_idx = [], max_idx = []])
```

- Compute histogram from given 1-channel image:

```python
>>> cv.CalcHist(image, hist[, accumulate, mask])
```

- ** Additional notes **

  (NB! Images with channels > 1, use Split()-function)

  Flesh tones are often easier to pick out after conversion to an HSV color space. It turns out that restricting ourselves to the hue and saturation planes is not only sufficient but also helps with recognition of flesh tones across ethnic groups. Sometimes normalized BGR works better than HSV in the context of lightning changes.

### Comparing 2 histograms

```python
>>> cv.CompareHist(hist1, hist2, method)
```
Methods

- **Correlation** - CV_COMP_CORREL, high score represents a better match than a low score.

\[ d_{correl}(H_1, H_2) = \frac{\sum_i H'_1(i) \cdot H'_2(i)}{\sqrt{\sum_i H'^2_1(i) \cdot H'^2_2(i)}} \]

- **Chi-square** - CV_COMP_CHISQR, low score represents a better match than a high score. A perfect match is 0 and a total mismatch is unbounded.

\[ d_{chisqr}(H_1, H_2) = \sum_i \frac{(H_1(i) - H_2(i))^2}{H_1(i) \cdot H_2(i)} \]

- **Intersection** - CV_COMP_INTERSECT, high score indicates good matches. If both histograms are normalized to 1, then a perfect match is 1 and total mismatch is 0.

\[ d_{chisqr}(H_1, H_2) = \sum_i \min(H_1(i), H_2(i)) \]

- **Bhattacharyya distance** - CV_COMP_BHATTACHARYYA, low scores indicate good matches and high scores indicate bad matches.

\[ d_{chisqr}(H_1, H_2) = \sqrt{1 - \sum_i \frac{\sqrt{H_1(i) \cdot H_2(i)}}{\sqrt{\sum_i H_1(i) \cdot \sum_i H_2(i)}}} \]

**PS:** intersection works well for quick-and-dirty matching and chiSquare or Bhattacharyya work best for slower but more accurate matches.

---

**Example: Calculate Histograms**

Compute histogram from given picture

```python
def compute_histogram(src, h_bins = 30, s_bins = 32, scale = 10):
    '''calculate histogram from picture'''
    #create images
    hsv = cv.CreateImage(cv.GetSize(src), 8, 3)
    hplane = cv.CreateImage(cv.GetSize(src), 8, 1)
    splane = cv.CreateImage(cv.GetSize(src), 8, 1)
    vplane = cv.CreateImage(cv.GetSize(src), 8, 1)
    planes = [hplane, splane]
    cv.CvtColor(src, hsv, cv.CV_BGR2HSV)
    cv.CvtPixToPlane(hsv, hplane, splane, vplane, None)
    #compute histogram
    hist = cv.CreateHist((h_bins, s_bins), cv.CV_HIST_ARRAY,
                        ranges = ((0, 180), (0, 255)), uniform = True)
    cv.CalcHist(planes, hist) #compute histogram
    cv.NormalizeHist(hist, 1.0) #normalize histo
    return hist
```

Show histogram
```python
def show_histogram(hist, hbins = 30, sbins = 32, scale = 10):
    # create image to use to visualize our histogram
    hist_img = cv.CreateImage((hbins*scale, sbins*scale), 8, 3)
    cv.Zero(hist_img)

    # populate our visualization with little squares
    vals = cv.GetMinMaxHistValue(hist)
    print "histogram vals: ", vals
    # TODO: try to vectorise this computation <-- python "for" is expensive
    for h in range(0, hbins):
        for s in range(0, sbins):
            bin_val = cv.QueryHistValue_2D(hist, h, s)
            intensity = cv.Round(bin_val * 255.0 / vals[1])
            p1 = (h * scale, s * scale)
            p2 = ((h-1)* scale, (s-1) * scale)
            color = cv.RGB(intensity, intensity, intensity)
            cv.Rectangle(hist_img, p1, p2, color, cv.CV_FILLED)

    # show image
    show_img("H-S histogram", hist_img)
```

Usage

```python
>>> src = cv.LoadImage("pictures/grid1.jpg", cv.CV_LOAD_IMAGE_COLOR)
>>> hist1 = compute_histogram(src)
>>> show_histogram(hist1, h_bins, s_bins, scale)
```
Earth mover’s distance (EMD)

Lighting changes can cause shifts in color values, although such shifts tend not to change the shape of the histogram of color values, but shift the color value locations and thus cause the previous histogram-matching schemes to fail.

EMD helps to match like histograms to like histograms even when the second histogram has shifted its been by looking for small distance measures.

It essentially measures how much work it would take to “shovel” one histogram shape into another, including moving part of the histogram to a new location. It works in any number of dimensions.

```python
>>> cv.CalcEMD2(signature1, signature2, distance_type[, distance_func, cost_matrix, ]
    flow, lower_bound, userdata)
```

- **signatures** - consist of rows containing the histogram bin count followed by its coordinates. If we had a bin in 3-dimensional histogram with a bin count of 537 at (x,y,z) index (7, 43, 11), then the signature row for that bin would be [537, 7;43,11]

- **distance_type**
  - CV_DIST_L1 = Manhattan distance
  - CV_DIST_L2 = Euclidean distance

Calculate signatures

```python
def compute_signatures(hist1, hist2, h_bins = 30, s_bins = 32):
    '''
demos how to convert 2 histograms into 2 signature
    '''
    num_rows = h_bins * s_bins
    sig1 = cv.CreateMat(num_rows, 3, cv.CV_32FC1)
    sig2 = cv.CreateMat(num_rows, 3, cv.CV_32FC1)
    #fill signatures
    #TODO: for production optimize this, use Numpy
    for h in range(0, h_bins):
        for s in range(0, s_bins):
            bin_val = cv.QueryHistValue_2D(hist1, h, s)
            cv.Set2D(sig1, h*s_bins + s, 0, bin_val) #bin value
            cv.Set2D(sig1, h*s_bins + s, 1, h) #coord1
            cv.Set2D(sig1, h*s_bins + s, 2, s) #coord2 #signature.2
            bin_val2 = cv.QueryHistValue_2D(hist2, h, s)
            cv.Set2D(sig2, h*s_bins + s, 0, bin_val2) #bin value
            cv.Set2D(sig2, h*s_bins + s, 1, h) #coord1
            cv.Set2D(sig2, h*s_bins + s, 2, s) #coord2

    return (sig1, sig2)
```

Calculate EMD

```python
def compute_emd(src1, src2, h_bins, s_bins, scale):
    hist1 = compute_histogram(src1, h_bins, s_bins, scale)
    hist2 = compute_histogram(src2, h_bins, s_bins, scale)
    sig1, sig2 = compute_signatures(hist1, hist2)
    emd = cv.CalcEMD2(sig1, sig2, cv.CV_DIST_L2)
    return emd
```

Usage

1.5. Earth mover’s distance (EMD)
Back projection

Back projection is a way of recording how well the pixels or patches of pixels fit the distribution of pixels in a histogram model. For example, if we have a histogram of flesh color then we can use back projection to find flesh color areas in an image.

```python
>>> cv.CalcBackProject(image, back_project, hist)
```

If the histogram is normalized, then `back_project(s)` value can be associated with conditional probability value, i.e the probability that a pixel in image is a member of the type characterized by the histogram in `hist`.

Little note:

If we know the overall probability of encountering a flesh-colored object in scene as well as the total probability of encountering of the range of flesh colors, then we can compute *p(F|C)* from *p(C|F)* by using well-known Bayes' rule (where F=prob(pixel is flesh) and C = prob(color of that pixel)):

```
p(F|C) = \frac{p(C|F) \times p(F)}{p(C)}
```

When `back_project` is a byte image rather than a float image, you should either not normalize the histogram or else scale it up before use. The reason is that the highest possible value in a normalized histogram is 1, so anything less than that be rounded down to 0, int the 8-bit image.

You might need to scale `back_project` in order to see the values with your eyes, depending on how high the values are in your histogram.

Patch-based back projection

```python
>>> cv.CalcBackProjectPatch(images, dst, patch_size, hist, method, factor)
```

Allows us to compute if a patch might contain a known object. It uses a sliding window over the entire input image. At each location in the input array of images, all the pixels in the patch are used to set one pixel in the destination image corresponding to center of patch. This is important because many properties of images such as gestures can not be determined at the level of individual pixels, but instead arise from groups of pixels.

Using patches, we can detect statistical properties that occur over local regions, such as the variations in local intensity that make up a texture up to the configuration of properties that make up a whole object.

Using local patches, there are 2 ways one might consider applying `CalcBackProjectPatch()`: as a region detector when the sampling window is smaller then the object and as an object when the sampling windows is the size of the object.

- `dst` - can be only a single-channel, floating-point image with size (images[0][0].width - patch_size.x +1, images[0][0].height - patch_size + 1)
- `method` - same methods types as used in `CompareHist()`.
- `factor` - is normalization level, also similar as used in `CompareHist()`
CHAPTER 2

Template matching

>>> cv.MatchTemplate(image, templ, result, method)

The method matches an actual image patch against an input image by “sliding” the patch over the input image using one of the matching methods described in this section.

- **image** - single 8-bit, floating-point plane or color images
- **templ** - patch from a similar image containing the object for which you are searching.
- **result** - single-channel byte or floating-point image of size, which can compute as we calculate dst argument in CalcBackProjectPatch documentation.
- **method** - is somewhat more complex, I = Input Image, T = the template and R = result
  - **square difference matching method**, cv.CV_TM_SQDIFF - that method match the squared difference, so perfect match will be 0 and bad matches will be large:
    \[
    R_{sq-diff}(x, y) = \sum_{x',y'} [T(x', y') - I(x + x', y + y')^2
    \]
  - **correlation matching method**, cv.CV_TM_CCORR - that method multiplicatively match the template against the image, so a perfect match will be large aand bad matches will be small or 0.
    \[
    R_{c-corr}(x, y) = \sum_{x',y'} [T(x', y') \times I(x + x', y + y')]^2
    \]
  - **correlation coefficient matching methods**, cv.CV_TM_CCOEFF - matches a template relative to its mean against the image relative to its mean, so a perfect mach will be 1 and a perfect mismatch will be -1; a value of 0 simly means random alignments.
    \[
    R_{c-coef}(x, y) = \sum_{x',y'} [T'(x', y') \times I'(x + x', y + y')]^2 T'(x', y') = T(x', y') -
    \]
\[
\frac{1}{(w \cdot h) \sum_{x',y'} T(x',y')} I'(x + x', y + y') - \frac{1}{(w \cdot h) \sum_{x',y'} I(x + x', y + y')}
\]

- **normalized methods** - for each of previous methods, there are also normalized versions. The normalized methods are useful, they can help reduce the effects of lightning differences between the template and the image. In each case, values are divided by factor \( Z \) to get normalized coefficients.

\[
Z(x,y) = \sqrt{\sum_{x',y'} T(x',y')^2 \times \sum_{x',y'} I(x + x', y + y')^2}
\]

Once we use cv.MatchTemplate() to obtain a matching result image we can then use cv.MinMaxLoc() to find the location of the best match.

**Example: template matching**

```python
def match_template(src, template):
# allocate output images
    results = []
    width = src.width - template.width + 1
    height = src.height - template.height + 1
    for i in range(0, 6):
        results.append(cv.CreateImage((width, height), 32, 1))

    # do the matching of the template with the image
    for i in range(0, 6):
        cv.MatchTemplate(src, template, results[i], i)

    # display results
    titles = ["SQ_DIFF", "SQ_DIFF_NORMED", "CCORR",
              "CCORR_NORMED", "CCOEFF", "CCOEFF_NORMED"]

    show_img("source", src)
    show_img("template", template)
    for i, title in enumerate(titles):
        show_img(title, results[i])
```

**Results**
• Source and template:

• SqDifference method:
• Correlation method:
• Correlation Coefficient method:
Although algorithms like the Canny edge detector can be used to find the edge pixels that separate different segments in an image, they do not tell anything about those edges as entities in themselves.

**Memory storage**

Memory storage is a method that handles memory allocation for dynamic objects. Memory storages are linked lists of Memory blocks that allow for fast allocation and de-allocation of continuous sets of blocks.

```python
>>> cv.CreateMemStorage(block_size = 0)
```

**Sequences**

(makes lot of sense in C, read original doc: link) One kind of object that can be stored inside a memory storage is a sequence. Sequences are themselves linked lists of other structures.

The sequence structure elements:

- **total** - returns how many elements are in sequence
- **h_prev, v_prev, h_next, v_next** - pointers that connects sequence items together.

in C: cvCreateSeq() - is not supported in Python library.

**Clear sequence**

```python
>>> cv.ClearSeq(seq)  #clears all elements of the sequence
```

**Direct access to sequence elements**

in C: cvGetSeqElem(seq, index) cvSeqElemIdx(seq, elem, block)

**Slices, copying, and moving data**
Contour Finding

A contour is a list of points that represent, in one way or another, a curve in an image. Contours are represented in OpenCV by sequences in which every entry in the Sequences encodes information about the location of the next point on the curve.

```python
>>> cv.FindContours(image, storage [,mode, method, offset])
```

Computes contours from binary images. It can take images created by Canny(), which have edge pixels in them, or images created by functions like Threshold(), AdaptiveThreshold(), in which the edges are implicit as boundaries between positive and negative regions.

- **image** - input image. NB! Uses input image as scratch space for computation, so if you need that image for anything later you should make a copy and pass that.
- **storage** - memory storage to keep results
- **mode** -
  - CV_RETR_EXTERNAL - retrieves only the extreme outer contours.
  - CV_RETR_LIST - retrieves all contours and puts them in the list.
  - CV_RETR_CCOMP - retrieves all the contours and organizes them into 2-level, where the top-level boundaries are external boundaries of the components and the second-level boundaries are boundaries of the holes.
  - CV_RETR_TREE - retrieves all the contours and reconstructs the full hierarchy of nested contours.
- **method**

  Next five values pertain to the the methods (i.e now the contours are approximated).
  - CV_CHAIN_CODE - output contours in the Freeman chain code
  - CV_CHAIN_APPROX_NONE - translates all the points from the chain code into points
  - CV_CHAIN_APPROX_SIMPLE - compresses horizontal, vertical, and diagonal segments, leaving only their ending points.
  - CV_CHAIN_APPROX_TC89_L1, CV_CHAIN_APPROX_TC89_KCOS - applies one of the flavors of the Teh-Chin chain approximation algorithm.
  - CV_LINK_RUNS - links horizontal segments of 1s; the only retrieval mode allowed by this method is CV_RETR_LIST.

```python
>>> cv.ApproxChains(src_seq, storage[, method, parameter, min_perimeter, recursive])
→#returns CvSeq
```

This function convert Freeman chains to polygonal representation with some approximation.

Freeman Chain Codes

With a Freeman chain, a polygon is represented as a sequence of steps in one of right directions; each step is designated by an integer from 0 to 7. Freeman chains have useful applications in recognition and other contexts.
Drawing Contours

```python
>>> drawContours(image, contours, contourIdx, color[, thickness[, lineType[, hierarchy[, maxLevel[, offset]]]]])
```

- **max_level** - tells how to handle any contours that might be attached to contour by means of the node tree variables. This argument can be set to indicate the maximum depth to traverse in the drawing.
  - max_level = 0 - means that all the contours on the same level as the input level (more exactly, the input contour and contours next to it) are drawn.
  - max_level = 1 - means that all the contours on the same as input and their children are drawn, and so forth.
  - If the contours in question were produced by FindContours() using either CV_RETR_CCOMP or CV_RETR_TREE mode, the the additional idiom of negative values for max_level is also supported. In this case, max_level = -1 is interpreted to mean that only the input contour will be drawn, max_level = -1 means that input contour and its direct children will the drawn, and so on.

- **offset** - contour will be drawn elsewhere than at the absolute coordinates by which it was defined. This feature is particularly useful when the contour has already been converted to center-of-mass or other local coordinates.

Polygon Approximation

If we are drawing a contour or are engaged in shape analysis, it is common to approximate a contour representing a polygon with another contour having fewer vertices.

```python
>>> curves = cv2.approxPolyDP(curve, epsilon, closed[, approxCurve])
```

- **epsilon** - precision parameter

Contour example

From drawings

3.4. Drawing Contours 17
```python
def main(from_file = False):
    #-- callback
    def update(level):
        dst = np.zeros((h,w,3), np.uint8)
        level = level - 3
        cv.drawContours(dst, contours, (-1, 3)[level <= 0], (128,255,255),
                        3, cv.CV_AA, hierarchy, abs(level))
        cv.imshow('contours', dst)
    #-- main content
    img = make_image()
    h, w = img.shape[:2]
    contours0, hierarchy = cv.findContours(img.copy(), cv.RETR_TREE, cv.CHAIN_APPROX_SIMPLE)
    contours = [cv.approxPolyDP(cnt, 3, True) for cnt in contours0]
    update(3)
    cv.createTrackbar("levels+3", "contours", 3, 21, update)
    cv.imshow('image', img)
    cv.waitKey()
```

Result

From file

Result

Chapter 3. Contours
Summary characteristics

length and area

```python
>>> cv2.arcLength(curve, closed) #-> retval
>>> cv2.contourArea(contour[, oriented]) #-> retval
```

Bounding boxes

```python
>>> cv2.boundingRect(points) #-> retval
```

Returns coordinates of bounding box

```python
>>> cv2.minAreaRect(points) #-> retval
```
Returns coordinates of bounding box with minimal area

**Enclosing circles and ellipses**

```python
>>> cv2.minEnclosingCircle(points) #-> center, radius
```

This routine works pretty much the same as the bounding box routines. Simply computes the smallest circle that completely encloses the given contour.

```python
>>> cv2.fitEllipse(points) #-> retval
```

Uses fitting function and return the ellipse that is the best approximation to the contour. This means that not all points in the contour will be enclosed in the ellipse returned. The fitting is done using a least-squares fitness function.

![Diagram](image)

*Figure 8-8. Ten-point contour with the minimal enclosing circle superimposed (a) and with the best-fitting ellipsoid (b); a box (c) is used by OpenCV to represent that ellipsoid*

**Matching Contours**

The most common task associated with contours is matching them in some way with one another.

**Matching with Moments**

One of the simplest ways to compare 2 contours. A moment is a gross characteristic of the contour computer by
integrating (summing) over all of the pixels of the contour. In general, we define the \((p,q)\) moment of contour as:

\[
\mu_{p,q} = \sum_{i=1}^{n} I(x, y) \cdot x^p \cdot y^q
\]

However, the moments resulting from that computation are not the best parameters for such comparisons in most practical cases. Use normalized moments, or Hu invariant moments.

*Hu invariant moments — a linear combination of the central moments. The idea here is that, by combining the different normalized central moments, it is possible to create an invariant function representing different aspects of the image in a way that is invariant to scale, rotation, and reflection.

```python
>>> cv2.matchShapes(contour1, contour2, method, parameter) #-> retval
```

Allows us to simply provide 2 contours and have their moments computed and compared according to a criterion that we provide.

- **method** - CV_CONTOURS_MATCH_I1, CV_CONTOURS_MATCH_I2, CV_CONTOURS_MATCH_I3

Hierarchical Matching

#TODO: seems its not supported yet.

### Contour convexity and convexity defects

Another useful way of comprehending the shape of an object or contour is to compute a convex hull for the object and then compute its convexity defects.
cv2.convexHull(points[, hull[, clockwise[, returnPoints]]]) #-> hull

- **points** - should be 32bit integers or floating numbers
- **clockwise** - either cv.CV_CLOCKWISE or CV_COUNTERCLOCKWISE, which will determine the orientation of the points when they are returned by the routine
- **returnPoints** - if True, then points themselves will be stored in the return array. If False, then only indices will be stored in the return array.

#TODO: find cv2 version >>> cv.ConvexityDefects(contour, convexhull, storage) #-> CvSeqOfCvConvexityDefect*

Computes defect and returns a sequence of the defects.
CHAPTER 4

Image parts and segmentation

Background subtraction

Because of its simplicity and because camera locations are fixed in many contexts, background subtraction (background differencing) is probably the most fundamental image processing operation for video security applications.

In order to perform background subtraction, we first must “learn” a model of the background. Once learned, this background model is compared against the current image and then the known background parts are subtracted away. The object left after subtraction are presumably new foreground objects.

Weaknesses of background subtraction

In order to take surrounding pixels into account, we could learn a multipart model, a simple example of which would be an extension of our basic independent pixel model to include a rudimentary sense of the brightness of neighboring pixels. In this case, we use the brightness of neighboring pixels to distinguish when neighboring pixel values are relatively bright or dim. We then learn effectively 2 models for the individual pixel: one for when the surrounding pixels are bright and one for when the surrounding pixels are dim. In this way we have a model that takes into account the surrounding context.

Scene modelling

In a higher-level “scene” model, we define multiple levels between foreground and Background states, and a timing-based method of slowly relegating unmoving foreground patches to background patches. In general, a scene model might contain multiple layers, from “new foreground” to older foreground on down to background.

There might also be some motion detection so that, when an object is moved, we can identify both its “positive aspect” (its new location) and its “negative” aspect (its old location, the “hole”).

Frame differencing

```python
>>> cv2.absdiff(src1, src2[, dst]) #-> dst
```
The very simplest background subtraction method is to subtract one frame from another (possibly several frames later) and then label any difference that is “big enough” the foreground.

Tends to catch the edges of moving objects.

For better and robust results, we need to clean up small noise areas; we might do this with Erode() or by using connected components. For color images, we could use the same code for each color channel and then combine the channels with cvOr().

**Averaging Background Method**

The averaging method basically learns the average and standard deviation of each pixel as its model of the background. It makes use of 3 OpenCV routines:

```python
>>> cv.accumulate(src, dst[, mask]) #-> None
```

To accumulate frame-to-frame image differences over time

```python
>>> cv.inRange(src, lowerb, upperb[, dst]) #-> dst
```

To segment the image (once a background model has been learned) into foreground and background regions;

```python
>>> np.logical_or(src1, src2) #not tested, maybe needs flatten and then
˓→packing pack old shape.
```

To compile segmentations from different color channels into a single mask image.

**Examples:**

```python
def average_background(capturer, show_images = False):
    ...,
    def accumulate_background(src, prev_img, first = False):
        ...,
        scratch = cv.cvtColor(src, 1)
        if first:
            cv.accumulate(scratch, avg_img)
            scratch = cv.absdiff(scratch, prev_img)
            cv.accumulate(scratch, diff_img)
        
        prev_img = scratch
        return prev_img

def set_high_threshold(scale):
    scratch = cv.convertScaleAbs(diff_img, None, scale)
    if scratch.dtype != np.float32:
        scratch = np.array(scratch, dtype = np.float32)
    highs = cv.add(scratch,avg_img)
    return cv.split(highs)

def set_low_threshold(scale):
    scratch = cv.convertScaleAbs(diff_img, None, scale)
    if scratch.dtype != np.float32:
        scratch = np.array(scratch, dtype = np.float32)
    lows = cv.subtract(avg_img, scratch)
    return cv.split(lows)
```
```python
def create_models(avg_img, diff_img):
    avg_img = cv.convertScaleAbs(avg_img, None, 1.0/count)
    diff_img = cv.convertScaleAbs(diff_img, None, 1.0/count)
    scalar = cv.cv.Scalar(1.0,1.0,1.0)
    temp = cv.cv.fromarray(diff_img)
    cv.cv.AddS(temp, scalar, temp)  #TODO: create decorator
    diff_img = np.asarray(temp)
    highs = set_high_threshold(7.0)
    lows = set_low_threshold(6.0)
    return (lows, highs)

def background_diff(src, mask):
    #scratch = cv.convertScaleAbs(src, None, 1)
    scratch = np.array(src, dtype = np.float32)  #to float
    grays = cv.split(scratch)
    #manage channels
    mask = cv.inRange(grays[0], lows[0], highs[0])
    maskt = cv.inRange(grays[1], lows[1], highs[1])
    mask = Or(mask, maskt)
    maskt = cv.inRange(grays[2], lows[2], highs[2])
    mask = Or(mask, maskt)
    mask = cv.cv.SubRS(mask, 255)
    return mask

avg_img = 0; diff_img = 0; scratch = 0
img = 0; mask = 0
count = 0.00001
while True:
    ret_val, img = capturer.read()
    print ret_val
    if ret_val == False:
        print "Didnt get value"
        break
    if count < 1:
        #initialize pictures
        height, width = img.shape[:2]
        avg_img, diff_img, scratch = map(create_empty_img,
                                         itertools.repeat((height, width, 3), 3))
        mask = create_empty_img((height, width, 1), dtype = np.int8)
        #init models
        #create_models(img, avg_img, diff_img)
        lows, highs = create_models(avg_img, diff_img)
        scratch = accumulate_background(img, scratch)
    else:
        scratch = accumulate_background(img, scratch)
        mask = background_diff(img, mask)
        if show_images:
            cv.imshow("background", scratch)
            cv.imshow("difference", diff_img)
            cv.imshow("average", avg_img)
        ch = cv.waitKey(100)
        if ch == 27 or count > 50:
            break
```

4.2. Scene modelling
Advanced background method

A nice method to deal scenes with varying lightning is to fit a time-series model to each pixel or group of pixels. The simplest way to form a codebook would be to compare a new value observed for a pixel with prior observed values. If the value is close to a prior value, then it is modeled as perturbation on that color. If it is not close, then it can see a new group of colors to be associated with that pixel. The result could be envisioned as a bunch of blob floating in RGB space, each blob representing a separate volume considered likely to be background.

In practise, the choice of RGB is not particularly optimal. It is almost always better to use a color space whose axis is aligned with brightness, such as the YUV color space. The reason for this is that, empirically, most of the variation in background tends to be along the brightness axis, not the color axis.

Codebook is made up of boxes that grow to cover common values seen over time. In the codebook method of learning a background model, each box is defined by 2 threshold (max, min) over each of the 3 color axes. These box boundary thresholds will expand (max getting larger, min getting smaller) if new background samples fall within a learning threshold above max or below min, respectively. If new background samples fall outside of the box and its learning thresholds, then a new box will be started.

examples:

#TODO

Comparing Background Methods

- The average difference method
  # Leaves behind a sloppier mask and breaks object into components.
- The codebook method
  # Can more accurately mode the fluctuations of the leaves and branches and so more precisely identify foreground object pixels from background pixels.
  # Yields less noise, also connected components can generate a fairly accurate object outline.

The Watershed algorithm

The Watershed algorithm converts lines in an image into “mountains” and uniform regions into “valleys” that can be used to help segment objects.

It 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 user-specified (or algorithm) points until these regions meet. Regions that merge across the marks so generated are segmented as belonging together as the image “fills up”

```python
>>> cv.watershed(image, markers) #-> None
```

#TODO: example
Image repair by Inpainting

Inpainting is method for removing damaged areas by taking the color and texture at the border of the damaged area and propagating and minxing it inside the damaged area.

It works well when provided the damaged area is not too “thick” and enough of the original texture and color remains around the boundaries of the damage.

```python
>>> cv.inpaint(src, inpaintMask, inpaintRange, flags[, dst]) #-> dst
```

- `flags` - cv.INPAINT_NS - Navier_stokes method, cv.INPAINT_TELEA - A.Telea’s method

#TODO: example

Mean-Shift segmentation

Pyramid segmentation uses a color merge (over a scale that depends on the similarity of the colors to one another) in order to segment images. This approach is based on minimizing the total energy in the image; here energy is defined by a link strength, which is further defined by color similarity.

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. As mean-shift windows move, all the points traversed by the windows that converge at a peak in the data become connected or “owned” by the peak. This ownership, radiating out from the densest peaks, forms the segmentation of the image.

```python
>>> cv.pyrMeanShiftFiltering(src, sp, sr[, dst[, maxLevel[, termcrit]]]) #-> dst
```

- `termcrit` - dictionary with keys, # type: cv.TERM_CRITERIA_COUNT | cv.TERM_CRITERIA_EPS | cv.TERM_CRITERIA_MAX_ITER
  # max_iter: integer
  # epsilon: double

#TODO: example

Delaunay Triangluation, Voronoi Tesselation

Delaunay triangluation is a technique for connecting points in a space into trianglular groups such that the minimum angle of all the angles in the triangulation is a maximum. This means that Delaunay triangulation tries to avoid long skinny triangles when triangulating points. For computational effieciency, the Delaunay algorithm invents a far-away outer bounding triangle from which the algorithm starts.

The gist of one of the more simple algorithms is as follows:

# Add the external triangle and start at one of its vertices ( this yields a definitive outer starting point)
# Add an internal point; then search over all the triangles’ circum-circles containing that point and remove those triangluations.
# Re-triangulate the graph, including the new point in the circum-circles of the just removed triangluations.
# Return to step 2 until there are no more points to add.
Voronoi tessellation is the dual image of the Delaunay triangulation, because the Delaunay lines define the distance between existing points and so the Voronoi lines “know” where they must intersect the Delaunay lines in order to keep equal distance between points.

These 2 methods, calculating the convex hull and nearest neighbor, are important basic operations for clustering and classifying points and point sets.

Figure 9-12. Delaunay triangulation: (a) set of points; (b) Delaunay triangulation of the point set with trailers to the outer bounding triangle; (c) example circles showing the circum-circle property

Figure 9-13. Voronoi tessellation, whereby all points within a given Voronoi cell are closer to their Delaunay point than to any other Delaunay point: (a) the Delaunay triangulation in bold with the corresponding Voronoi tessellation in fine lines; (b) the Voronoi cells around each Delaunay point

#TODO: examples
CHAPTER 5

Tracking and motion

**Identification** amounts to finding the object of interest from one frame in a subsequent frame of the video stream. **Tracking** things that we have not yet identified is a related problem. Tracking unidentified objects is important when we wish to determine what is interesting based on its motion — or when an object’s motion is precisely what makes it interesting.

**Corner finding**

If we choose a point that is unique then we have a pretty good chance of finding that point again. In practice, the point or feature we select should be unique, or nearly unique, and should be parameterizable in such a way that it can be compared to other points in another image.

![Figure 10-1. The points in circles are good points to track, whereas those in boxes—even sharply defined edges—are poor choices](image)

Intuitively, corners — not edges — are the points that contain enough information to be picked out from 1 frame to the next. The most commonly used definition of a corner was provided by Harris. This definition relies on the matrix of the second-order derivatives ($\Psi^2 x$, $\Psi^2 y$, $\Psi x \Psi y$) of the image intensities.
We can think of the second-order derivatives of images, taken at all points in the image, as forming new “second-derivative images” or, when combined together, a new Hessian image.

Corners, by Harris’s definition, are places in the image where the autocorrelation matrix of the second derivatives has 2 large eigenvalues. In essence this means that there is texture (or edges) going in at least 2 separate directions centered around such a point, just as real corners have at least 2 edges meeting in a point.

\[
M(x,y) = \begin{bmatrix}
\sum_{-K \leq i,j \leq K} w_{ij} I_x^2(x+i,y+j) & \sum_{-K \leq i,j \leq K} w_{ij} I_x(x+i,y+j)I_y(x+i,y+j) \\
\sum_{-K \leq i,j \leq K} w_{ij} I_x(x+i,y+j)I_y(x+i,y+j) & \sum_{-K \leq i,j \leq K} w_{ij} I_y^2(x+i,y+j)
\end{bmatrix}
\]

Second derivatives are useful because they do not respond to uniform gradients ~ invariant to rotation.

```python
>>> cv.goodFeaturesToTrack(image, maxCorners, qualityLevel, minDistance [], corners, mask[, blockSize[, useHarrisDetector[, k]]]]) #-> corners
```

This routine implements the Shi and Tomasi definition. This function conveniently computes the second derivatives (using the Sobel operators) that are needed and from those computes the needed eigenvalues. It then returns a list of points that meet our definition of being good for tracking.

### Subpixel corners

Subpixel corner locations are a common measurement used in camera calibration or when tracking to reconstruct the camera’s path or the three dimensional structure of a tracked object.

The trick for refining corner locations to subpixel accuracy: We use the mathematical fact that dot product between a vector and an orthogonal vector is 0; this situation occurs at corner locations, as shown above:

![Subpixel corners](image)

*Figure 10-2. Finding corners to subpixel accuracy: (a) the image area around the point \( p \) is uniform and so its gradient is 0; (b) the gradient at the edge is orthogonal to the vector \( q-p \) along the edge; in either case, the dot product between the gradient at \( p \) and the vector \( q-p \) is 0 (see text)*
### Invariant features

SIFT (*scale-invariant feature transform*) detect the dominant gradient orientation at its locations and records its local gradient histogram results with respect to this orientation. SIFT is also rotationally invariant. As a result, SIFT features are relatively well behaved under small affine transformations.

#TODO: examples

### Optical Flow

We can associate some kind of velocity with each pixel in the frame or, equivalently, some displacement that represents the distance a pixel has moved between the previous frame and the current frame. Such a construction is usually referred to as a dense optical flow, which associates a velocity with every pixel in an image.

- **Lucas-Kanade method** The LK algorithm can be applied in sparse context because it relies only on local information that is derived from small window surrounding each of the points of interest. The disadvantage of using small local windows in LK is that large motions can move points outside of the local window and thus become impossible for the algorithm to find. (leads to pyramidal LK algorithm). The basic idea of the LK algorithm rests on 3 assumptions:
  
  # Brightness constancy - A pixel from the image of an object in the sscene does not change in appearance as it moves from frame to frame.
  
  # Temporal persistence or small movements - The image motion of a surface patch changes slowly in time. In practise, this means the temporal increments are fast enough relative to the scale of motion in the image that the object does not move much from frame to frame.
  
  # Spatial coherence - Neighboring points in a scene belong to the same surface, have similar motion, and project to nearby points on the image plane.

The recommended technique is first to solve for optical flow at the top layer and then to use the resulting motion estimates as the starting point for the next layerdown. We continue going down the pyramid in this manner until we reach the lowest level. Thus we minimize the violations of our motion assumptions and so can track faster and longer motions.

```python
>>> cv.calcOpticalFlowPyrLK(prevImg, nextImg, prevPts[, nextPts[, status[, err[, winSize[, maxLevel[, criteria[, derivLambda[, flags[, minEigThreshold]]]]]]]]]) #-> nextPts, status, err
```

#TODO: examples

### Dense Tracking Techniques

- **Horn-Schunk** method

```python
>>> cv.cv.CalcOpticalFlowHS(prev, curr, usePrevious, velx, vely, lambda, ) #-> None
```

#TODO: examples

5.3. Invariant features 31
The parameter \( \lambda \) is weight related to the Lagrange multiplier, which arises when we attempt to minimize simultaneously both the motion-Brightness equation and the smoothness equations; it represents the relative weight given to the errors in each as we minimize.

**Block matching method**

The term “block matching” is a catchall for a whole class of similar algorithms in which the image is divided into small regions called blocks. Blocks are typically squares and contain some number of pixels. These blocks may overlap and, in practice, often do.

Block-matching algorithms attempt to divide both the previous and current images into blocks and then compute the motion of these blocks.

Because block-matching algorithms operate on aggregates of pixels, not on individual pixels, the returned “velocity images” are typically of lower resolution than the input images. This is not always the case; it depends on the severity of the overlap between the blocks; The size of the result images is given by the following formula:

\[
W_{\text{result}} = \left( \frac{W_{\text{prev}} - W_{\text{block}} + W_{\text{shiftsize}}}{W_{\text{shiftsize}}} \right) \text{floor}
\]

\[
H_{\text{result}} = \left( \frac{H_{\text{prev}} - H_{\text{block}} + H_{\text{shiftsize}}}{H_{\text{shiftsize}}} \right) \text{floor}
\]

The Kalman Filter

The basic idea behind the Kalman filter is that, under a strong but reasonable set of assumptions, it will be possible to build a model for the state of the system that maximizes the posteriori probability of those previous measurements.

There are 3 important assumptions required in the theoretical construction of the Kalman filter:

- the system being modeled is linear - the state of the system at time \( k \) can be modeled as some matrix multiplied by the state at time \( k-1 \)
- the noise that measurements are subject to is “white” - aka random
- the noise is also Gaussian in nature.

**Some Kalman math**

The new mean value :math: \( \bar{x}_{12} \) is just a weighted combination of 2 measured means, where the weighting is determined by the relative uncertainties of the 2 measurements. \( \bar{x}_{12} = \left( \frac{\sigma^2_2}{\sigma^2_1 + \sigma^2_2} \right) x_1 + \left( \frac{\sigma^2_1}{\sigma^2_1 + \sigma^2_2} \right) x_2 \) With the new mean :math: \( \bar{x}_{12} \), we can substitute this value into our expression \( p_{12}(x) \) and after substantial rearranging, identify the uncertainty as:

\[
\sigma^2_{12} = \frac{\sigma^2_1 \sigma^2_2}{\sigma^2_1 + \sigma^2_2}
\]

This formula says that when we make a new measurement with a new mean and uncertainty, we can combine that measurement with the mean and uncertainty we already have to obtain a new state that is characterized by a still newer mean and uncertainty.

This property that 2 Gaussian measurements, when combined, are equivalent to a single Gaussian measurement (with a computable mean and uncertainty) will be the most important feature for us. It means that when we have M measurements, we can combine the first 2, then the third with the combination of the first 2, then the fourth with the combination of the first three and so on. This is what happens with tracking in computer vision; we obtain one measure followed by another.

The new information :math: \( (x_{2} - \hat{x}_{1}) \), seen at time step 2, is called the innovation. Optimal iterative update factor \( K \) and following convenient recursion form:
Systems with dynamics

During the prediction phase, we use what we know to figure out where we expect the system to be before we attempt to integrate a new measurement.

In the context of Kalman filters, there are 3 kinds if motion that we would like to consider.

• dynamical motion = motion that we expect as a direct result of the state of the system when last we measured it.
• control motion - motion that we expect because some external influence applied to the system of which, for whatever reason, we happen to be aware.
• random motion - includes any motion that are not known or not under our control. The effect of such random motion will be to simply increase the variance of our state estimates with the passage of time.

To include dynamics in our simulation model, we would first do an “update” step before including a new measurement. This update step would include first applying any knowledge we have about the motion of the object according to its prior state, applying any additional information resulting from actions that we ourselfe have taken or that we know to have been taken on the system from another outside agent, and finally, incorporating our notion of random events that might have changed the state of the system since we last measured it. Once those factors have been applied, we can then incorporate our next new measurement.

We will generalize the description of the state at time step k to be the following function:

\[
\begin{align*}
x'_{k} &= Fx_{k-1} + Bu_{k} + w_{k} \\
z_{k} &= Hkx_{k} + v_{k}
\end{align*}
\]

* \(x'_{k}\) - \(n \times n\)-dimensional vector of state components
* \(F\) - \(n\)-by-\(n\) transfer matrix
* \(B\) - \(n\)-by-\(c\) matrix that related these control inputs to the state change/
* \(u_{k}\) - \(isnew\)
* \(w_{k}\) - \(processnoise\)
* \(H_{k}\) - \(m\)-by-\(n\)matrix
* \(v_{k}\) - \(measurementerror\)
Now we can optimally compute the updated values for \(x_{k}\) and \(P_{k}\) when new measurement is available:

\[
\begin{align*}
x_{k} &= x'_{k} + K(z'_{k} - Hkx'_{k}) \\
P_{k} &= (I - KkH_{k}) \times P'_{k}
\end{align*}
\]

The Kalman filter models a single hypothesis. Because the underlying model of the probability distribution for that hypothesis is unimodal Gaussian, it is not possible to represent multiple hypotheses simultaneously using the Kalman filter.

#TODO example:
The Condensation Algorithm

A somewhat more advanced technique, which is based on broader class of estimators called particle filters.

#TODO: example
The Kalman filter is a set of mathematical equations that provides an efficient computational (recursive) means to estimate the state of a process, in a way that minimizes the mean of the squared error.

**Mantra**

A Kalman filter is a computational algorithm that processes measurements to deduce a minimum variance, unbiased error estimate of the state of a system by utilizing:

- knowledge of system
- measurement dynamics
- assumed statistics of system noises
- measurement errors
- initial condition information

**The Discrete Kalman Filter**
CHAPTER 7

Indices and tables

- genindex
- modindex
- search