



Motivation

OpenCV is quite well known in the domain of image manipulation. It has all those secret boots you can wear and use to apply filters, resize, rotate, find shapes, remove background etc ...

In those days of neural networks, image pre-analysis is actual a fundamental of preparing a training set.

|

Short Sample

Welcome to this tutorial. This is a short sample to show how to use this fantastic API to do OpenCV stuff in the browser.

```
[ns composed-pine
  (:require
    [opencv3.utils :as u]
    [opencv3.core :refer :all]))
```

```
nil
```

```
(defn load-cvt-resize-show[url]
  (-> url
        (u/mat-from-url)
        (cvt-color! COLOR_RGB2GRAY)
        (resize! (new-size 150 200))
        (u/mat-view)))
```

```
 #'composed-pine/load-cvt-resize-show
```



Interactive OpenCV Introduction

Manipulate images with clojure and opencv

```
(ns spacial-dusk
  (:require
    [opencv3.core :refer :all]
    [opencv3.utils :as u]))
```

```
nil
```

```
(def neko
  (u/mat-from-url "https://s-media-cache-
ak0.pinimg.com/236x/10/2c/75/102c756d7e808deff666f3edf540abba.jpg" ))
(u/mat-view neko)
```



```
(def gray (new-mat))
(cvt-color neko gray COLOR_RGB2GRAY)
(u/mat-view gray)
```



```
(def small-gray-neko
  (u/resize-by gray 0.5))
(u/mat-view small-gray-neko)
```



```
(def equalized (new-mat))
(equalize-hist small-gray-neko equalized)
(u/mat-view equalized)
```

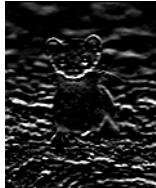


```
(def dilated (u/mat-from small-gray-neko))
(def dilation-size 2)
(def element (get-structuring-element MORPH_RECT
  (new-size (inc (* 2 dilation-size)) (inc (* 2 dilation-size)))))  

(dilate small-gray-neko dilated element)
(u/mat-view dilated)
```



```
(def sobelled (u/mat-from small-gray-neko))
(sobel small-gray-neko sobelled -1 0 1)
(u/mat-view sobelled)
```



```
(def bit_not (u/mat-from small-gray-neko)_
(bitwise-not small-gray-neko bit_not)
(u/mat-view bit_not)
```



Color Maps

This short tutorial will show you how to change the color maps of loaded pictures. A colormap is defined by a m-by-3 matrix of real numbers between 0.0 and 1.0. Each row is an RGB vector that defines one color.

This follows the story of kutsushita, the cat that likes to change color.

```
; first we import the usual namespaces.
(ns affectionate-thorns
  (:require
    [opencv3.utils :as u]
    [opencv3.colors.rgb :as color]
    [opencv3.core :refer :all]))
```

```
nil
```

Changing color space for a default color space is done through the function **apply-color-map!** from opencv3.core. The mat as well as the colormap id is passed as parameter.

Using the chaining method we can apply the BONE color map using the code below:

```
(-> "resources/images/cats/onsofa.jpg"
  imread
  (u/resize-by 0.5)
  (apply-color-map! COLORMAP_BONE)
  (u/mat-view))
```



You just saw one colormap, but a few other default color maps are available by default. The array below is a list of all the available color maps through OpenCV core.

```
(def colors-maps
[ "COLORMAP_HOT"
  "COLORMAP_HSV"
  "COLORMAP_JET"
  "COLORMAP_BONE"
  "COLORMAP_COOL"
  "COLORMAP_PINK"
  "COLORMAP_RAINBOW"
  "COLORMAP_OCEAN"
  "COLORMAP_WINTER"
  "COLORMAP_SUMMER"
  "COLORMAP_AUTUMN"
  "COLORMAP_SPRING" ])
```

```
#'affectionate-thorns/colors-maps
```

```
(defn watermark [ source text ]
  (put-text! source
  text
  (new-point 30 30) FONT_HERSHEY_PLAIN 1 (new-scalar 255 255 255) 2))

(defn change-color [ source color-map-string ]
  (->
  source
  clone
  (apply-color-map! (eval (read-string (str "opencv3.core/" color-map-string)))) 
  (watermark color-map-string)))

(defn apply-all-colormaps [ source ]
  (cons
    (-> source clone (watermark "ORIGINAL"))
    (map (partial change-color source) colors-maps))) )
```

```
#'affectionate-thorns/apply-all-colormaps
```

```
(def source
  (-> "http://sites.psu.edu/siowfa15/wp-content/uploads/sites/29639/2015/10/cat.jpg"
       (u/mat-from-url)
       (u/resize-by 0.1)))
```

```
#'affectionate-thorns/source
```

```
(def targets (apply-all-colormaps source))
(u/mat-view (vconcat! targets))
```



Defining your own color space is also possible using the function **transform!**. transform needs a matrix doing the mapping from rgb to some mapping. The first line of the matrix is the blue value, the second line is the green value, the third value is the red value.

The below matrix makes so that all red pixels are turned to green.

```
(def custom (u/matrix-to-mat [  
  [1 0 0] ; blue  
  [0 1 0] ; green  
  [0 1 0] ; red  
]))  
(-> img  
  clone  
  (transform! custom)  
  (u/mat-view ))
```



You could use the same technique to define your own sepia filter.

```
(def img (-> "resources/images/cats/onsofa.jpg" imread (u/resize-by 0.5)))  
  
(def blue-sepia (u/matrix-to-mat [  
  [0.393 0.769 0.189] ; blue  
  [0.349 0.686 0.168] ; green  
  [0.272 0.534 0.131] ; red  
]))  
(-> img  
  clone  
  (transform! blue-sepia)  
  (u/mat-view ))
```



And a regular sepia is done using a similar matrix.

```
(def sepia-2 (u/matrix-to-mat [  
  [0.131 0.534 0.272]  
  [0.168 0.686 0.349]  
  [0.189 0.769 0.393]  
]))  
(-> img  
  clone  
  (transform! sepia-2)  
  (u/mat-view ))
```



Cartoon Cats

Because we all love cats and cartoon ...

```
(ns opencv3.cartoon2
  (:require
    [opencv3.core :refer :all]
    [opencv3.utils :as u]))
```

```
nil
```

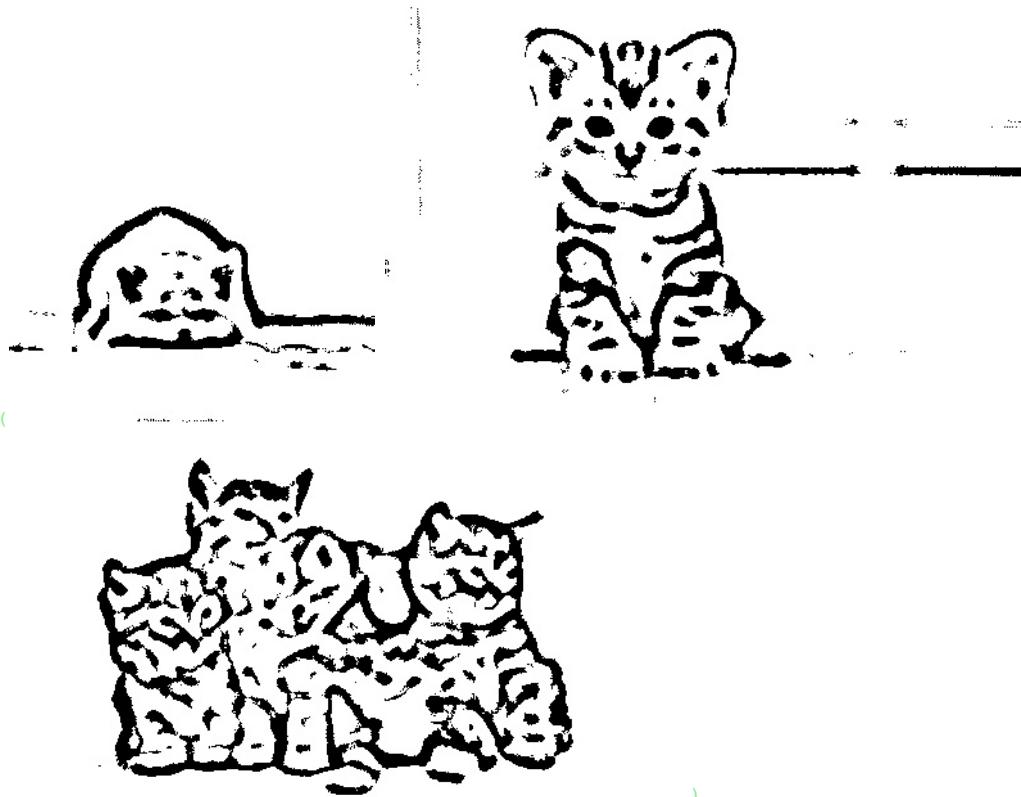
```
(defn cartoon
  [buffer]
  (->
    (cvt-color! buffer COLOR_RGB2GRAY))
  (let [ c (clone buffer)]
    ; (bilateral-filter buffer c 9 9 7)
    (bilateral-filter buffer c 10 250 50)
    (-> c
      (median-blur! 7)
      (adaptive-threshold! 255 ADAPTIVE_THRESH_MEAN_C THRESH_BINARY 9 3)
      (cvt-color! COLOR_GRAY2RGB))))
```

```
(defn cartoon-me[url]
```

```
(-> url
  u/mat-from-url
  (u/resize-by 0.5)
  cartoon
  u/mat-view))
```

```
#'opencv3.cartoon2/cartoon-me
```

```
(map
  cartoon-me
  [ "http://www.petmd.com/sites/default/files/sleepy-cat-125522297.jpg"
    "http://www.readersdigest.ca/wp-content/uploads/2011/01/4-ways-cheer-up-depressed-
cat.jpg"
    "https://hyatoky.com/wp-content/uploads/cute-cat-wallpapers-hd-1080x580.jpg"])
```



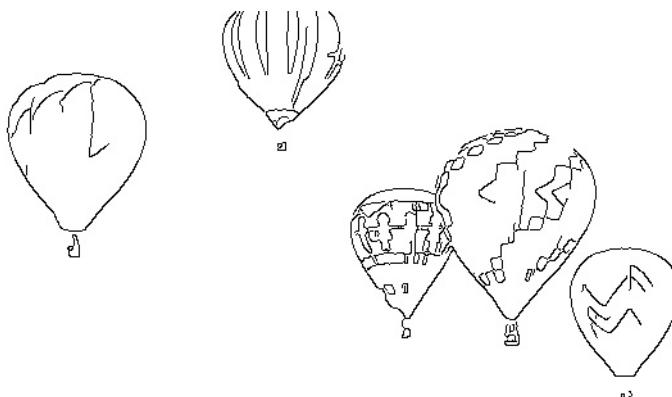
More Api Examples

We have seen very simple syntax usage, now its time to make use of those pipelines.

```
(ns opencv3.dolphin
  (:require
    [opencv3.utils :as u]
    [opencv3.core :refer :all]))
```

```
nil
```

```
(-> "http://www.v3wall.com/wallpaper/1366_768/0912/1366_768_20091223010850201138.jpg"
u/matrix-from-url
(u/resize-by 0.4)
(median-blur! 3)
(cvt-color! COLOR_BGR2GRAY)
(canny! 300.0 100.0 3 true)
(bitwise-not!)
(u/matrix-view))
```



```
(-> "resources/images/cat.jpg"
imread
(clone)
(u/resize-by 0.3)
(median-blur! 7)
(cvt-color! COLOR_BGR2GRAY)
bitwise-not!
u/matrix-view)
```

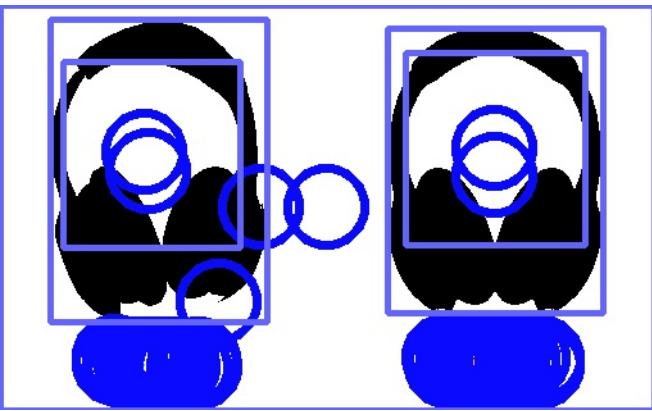


```
(def headphones (->
  "resources/morph/headphone.png"
  imread
  clone
  (cvt-color! COLOR_BGR2GRAY))

(def contours (new-arraylist))
(find-contours headphones contours (new-mat) RETR_LIST CHAIN_APPROX_SIMPLE)

(doseq [c contours]
  (if (> (contour-area c) 100)
    (let [ rect (bounding-rect c)]
      (if (> (.height rect) 28)
        (rectangle
          image-c
          (new-point (.x rect) (.y rect))
          (new-point (+ (.width rect) (.x rect)) (+ (.y rect) (.height rect)))
          (new-scalar 255 100 100)
          3))))))
```

(u/mat-view image-c)



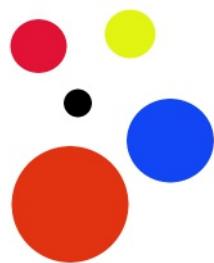
Find shapes with hough circles

This tutorial shows how to find shape with a given color within an image.

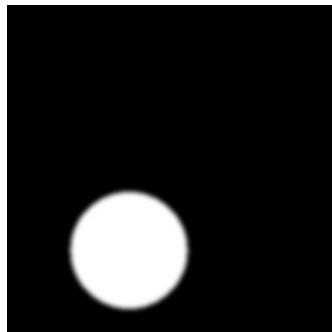
```
(ns combative-atoll
  (:require
    [opencv3.core :refer :all]
    [opencv3.utils :as u]))
```

nil

```
(def bgr-image
  (-> "resources/detect/circles.jpg" imread (u/resize-by 0.5) ))
(u/mat-view bgr-image)
```

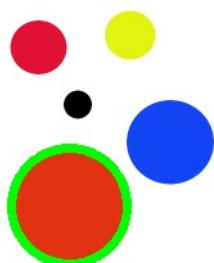


```
(def ogr-image
  (-> bgr-image
    (clone)
    (median-blur! 3)
    (cvt-color! COLOR_BGR2HSV)
    (in-range! (new-scalar 0 100 100) (new-scalar 10 255 255))
    (gaussian-blur! (new-size 9 9) 2 2)))
(u/mat-view ogr-image)
```



Let's find the circles using the **hough-circles** function from opencv core.

```
(def circles (new-mat)
  (hough-circles ogr-image circles CV_HOUGH_GRADIENT 1 (/ (.rows bgr-image) 8) 100 20 0 0)
  (dotimes [i (.cols circles)]
    (let [ circle (.get circles 0 i) x (nth circle 0) y (nth circle 1) r (nth circle 2) p
      (new-point x y)]
      (opencv3.core/circle bgr-image p (int r) (new-scalar 0 255 0) 5)))
  (u/mat-view bgr-image))
```



Using hough techniques to find lines and circles

Here we learn how to use opencv's hough lines

```
(ns opencv3.hough
  (:require
    [opencv3.utils :as u]
    [opencv3.colors.rgb :as color]
    [opencv3.core :refer :all]))
```

```
nil
```

Let's load the target image, and convert it to gray with a bit of blur, so that canny can work on it.

```
(def parking (-> "resources/images/lines/parking.png" imread (u/resize-by 0.3)))
(u/mat-view parking)
```

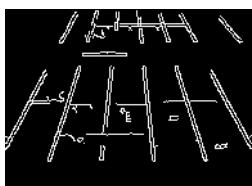


```
(def gray      (-> parking clone (cvt-color! COLOR_BGR2GRAY) (gaussian-blur! (new-size 3
3) 0 ) ))
(u/mat-view gray)
```



Edges detection is done using canny.

```
(def edges      (-> gray clone (canny! 50 150 )))
(u/mat-view edges)
```



Let's setup the parameters for the hough lines detection.

```
(def rho 1) ; distance resolution in pixels of the Hough grid
(def theta (/ Math/PI 180)) ; # angular resolution in radians of the Hough grid
(def min-intersections 15) ; # minimum number of votes (intersections in Hough grid
cell)
(def min-line-length 50) ; # minimum number of pixels making up a line
(def max-line-gap 20) ; # maximum gap in pixels between connectable line segments

(def lines (new-mat))
(hough-lines-p edges lines rho theta min-intersections min-line-length max-line-gap)
```

```
nil
```

Now that we have the lines, let's draw them on a clone of the original picture.

```
(def result (clone parking))
(dotimes [ i (.rows lines)]
(let [ val (.get lines i 0)]
  (line result
    (new-point (nth val 0) (nth val 1))
    (new-point (nth val 2) (nth val 3)))
  color/teal)
```

```
(2))  
(u/mat-view result)
```



Finally, output all the intermediate images used in this guide. Let's not forget that hconcat and vconcat need all the pictures to be in the same format, in our case, the same number of channels. We convert back *gray* and *edges* with the proper number of channels.

```
(u/mat-view (hconcat! [  
  parking  
  (-> gray clone (cvt-color! COLOR_GRAY2RGB))  
  (-> edges clone (cvt-color! COLOR_GRAY2RGB))  
  result] ))
```



```
(def pool  
  (->  
    "https://raw.githubusercontent.com/badlogic/opencv-fun/master/data/topdown-6.jpg"  
    u/mat-from-url  
    (u/resize-by 0.5)))  
(u/mat-view pool)
```



```
(def gray (-> pool clone (cvt-color! COLOR_BGR2GRAY)))  
(def minRadius 14)  
(def maxRadius 18)  
(def circles (new-mat))  
(hough-circles gray circles CV_HOUGH_GRADIENT 1 minRadius 120 10 minRadius maxRadius)
```

```
nil
```

```
(def output (clone pool))  
(dotimes [i (.cols circles)]  
  (let [ circle (.get circles 0 i) x (nth circle 0) y (nth circle 1) r (nth circle 2) p  
        (new-point x y)]  
    (opencv3.core/circle output p (int r) color/red-2 1)))  
(u/mat-view output)
```



Simple Foreground Background Diff

Here we have a background picture and we would like to discover new objects on top of this background.

We will use the simple opencv3.core function **absdiff** and then apply a simple and large **threshold** to pick up all the new objects.

The reference clojure code file can be found here: [bgdiff.clj](#)

Let's see first what our background looks like.

```
(ns scenic-iceberg
  (:require
    [opencv3.core :refer :all]
    [opencv3.utils :as u]))
(def bg (-> "resources/images/bgdiff/header.png" imread (u/resize-by 0.5)))
(u/mat-view bg)
```



Then put our hand in front of that background.

```
(def fg (-> "resources/images/bgdiff/front.png" imread (u/resize-by 0.5)))
(u/mat-view fg)
```



The **absdiff** function from OpenCV core is used, and we can directly see the output.

```
(def output (new-mat))
(absdiff bg fg output)
(u/mat-view output)
```



```
; diff in color
(def fg-1
  (-> output
    clone
    (threshold! 10 255 1)))
(u/mat-view fg-1)
```



You can then use a very permissive threshold to find out relevant shapes.

```
; diff in gray
(def fg-2
  (-> output
    clone
    (cvt-color! COLOR_RGB2GRAY)
    (threshold! 10 255 1)))
(u/mat-view fg-2)
```



Blur Detection

Is my image blurred ?

```

(ns wealthy-darkness
  (:require
    [opencv3.core :refer :all]
    [opencv3.colors.rgb :as color]
    [opencv3.utils :as u]))

; laplacian variation
;
; https://stackoverflow.com/questions/36413394/opencv-variation-of-the-laplacian-java
; http://www.pyimagesearch.com/2015/09/07/blur-detection-with-opencv/
; used to detect blur in an image

(def img
  (-> "resources/images/cat.jpg" imread))

(def kernel
  (u/matrix-to-mat
  [ [ 0 -1 0 ]
    [ -1 4 -1 ]
    [ 0 -1 0 ]]))

(filter-2-d! img -1 kernel)
(def std (new-matofdouble))
(def median (new-matofdouble))
(mean-std-dev img median std)

(Math/pow (first (.get std 0 0)) 2)

; implementation using a function
;

(def laplacian-kernel (u/matrix-to-mat
[ [ 0 -1 0 ]
  [ -1 4 -1 ]
  [ 0 -1 0 ]]))

(defn std-laplacian [img]
  (let [ std (new-matofdouble)]
    (filter-2-d! img -1 laplacian-kernel)
    (mean-std-dev img (new-matofdouble) std)
    (Math/pow (first (.get std 0 0)) 2)))

(defn is-image-blurred? [img]
  (< (std-laplacian (clone img)) 100))

(defn mark-blurred! [ __img ]
  (let [ _text (if (is-image-blurred? __img) "BLUR" "STILL") ]
    (put-text __img _text (new-point 30 30) FONT_ITALIC 1.0 color/blue-2 2)
    __img))

#'wealthy-darkness/mark-blurred!

```

```

(map #(-> % imread (u/resize-by 0.5) mark-blurred! u/mat-view)
  ["resources/images/tiger-blur.gif"
   "resources/blurred/blurred_cat.jpg"
   "resources/nico.jpg"])

```





Tennis Ball

This tutorial proposes options to find a tennis ball in a picture using two different techniques. This will also play with parameters, to show how to avoid finding false positive, i.e objects that are nowhere near the one we are looking for.

First, let's load the image that will be the base for our image finding exercice.

```
ns divine-briars
(:require
[opencv3.utils :as u]
[opencv3.colors.rgb :as color]
[opencv3.core :refer :all]))
```

```
nil
```

```
(def img (-> "http://i.imgur.com/uONRu60.jpg" (u/mat-from-url) (u/resize-by 0.5)))
(u/mat-view img)
```



Using Hough Circles

```
(def hsv (-> img clone (cvt-color! COLOR_RGB2HSV)))
(def thresh-image (new-mat))
(in-range hsv (new-scalar 50 100 0) (new-scalar 95 255 255) thresh-image)
; the ball is here
(u/mat-view thresh-image)
```



```
(let[ circles (new-mat) output (clone img) minRadius 25 maxRadius 40 ]
(hough-circles thresh-image circles CV_HOUGH_GRADIENT 1 minRadius 120 15 minRadius
maxRadius)
(dotimes [i (.cols circles)]
(let [ circle (.get circles 0 i) x (nth circle 0) y (nth circle 1) r (nth circle 2) p
(new-point x y)]
(opencv3.core/circle output p (int r) color/plum 2)))
; the ball is detected
(u/mat-view output))
```



Method2: Using Find Contours

Find object with find contours gives less chance to parameters, the **find-contours** function from core does pretty much all for us.

```
(let[ hsv (-> img clone (cvt-color! COLOR_RGB2HSV))
      thresh-image (new-mat)
      contours (new-arraylist)
      output (clone img)]
  (in-range hsv
             (new-scalar 50 100 0)
             (new-scalar 95 255 255)
             thresh-image)

  (find-contours
   thresh-image
   contours
   (new-mat) ; mask
   RETR_LIST
   CHAIN_APPROX_SIMPLE)

  (dotimes [ci (.size contours)]
    (if (> (contour-area (.get contours ci)) 100 )
        (draw-contours output contours ci color/plum FILLED)))
  (u/mat-view output))
```

