redCV Open Source Computer Vision Library

Code Sample Documentation

©Fyodor Shchukin

François Jouen
Human and Artificial Cognitions Laboratory
Ecole Pratique des Hautes Etudes
Université Paris Sciences et Lettres

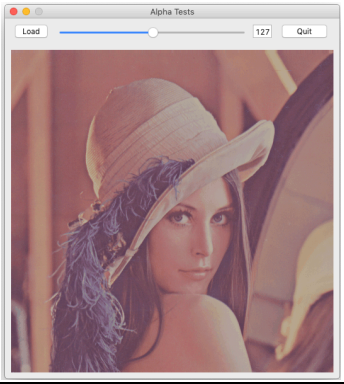
redCV is around 450 functions, that can be used for basic, but fast image processing with Red language. You'll find in *redCV/samples* directory a lot of code illustrating how Red language is performant for mathematics and image processing. Samples are organized by categories of image processing methods. Under progress. Enjoy☺
You'll find also some extra documentation here: http://redlcv.blogspot.com

© Red Foundation, 2019-2020

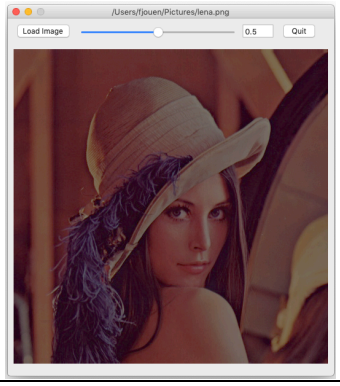
redCV/samples/image_alpha

The idea here is to illustrate how we can play with *argb* values to mix images or to control intensity for nice rendering. The a (transparency) channel varies from 0 (opaque) to 255 (transparent).

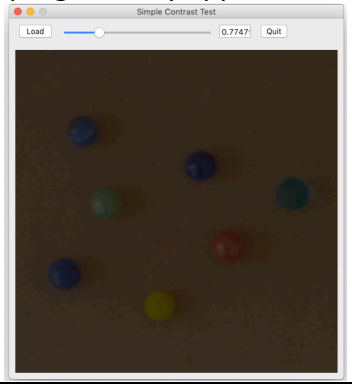
transparency.red: calls *rcvSetAlpha*. This code is trivial and just uses an integer value between 0 and 255 to control image transparency.



Intensity.red : calls *rcvSetIntensity*. This code is similar but uses a float! value between 0.0 and 255.0 for an accurate rendering.



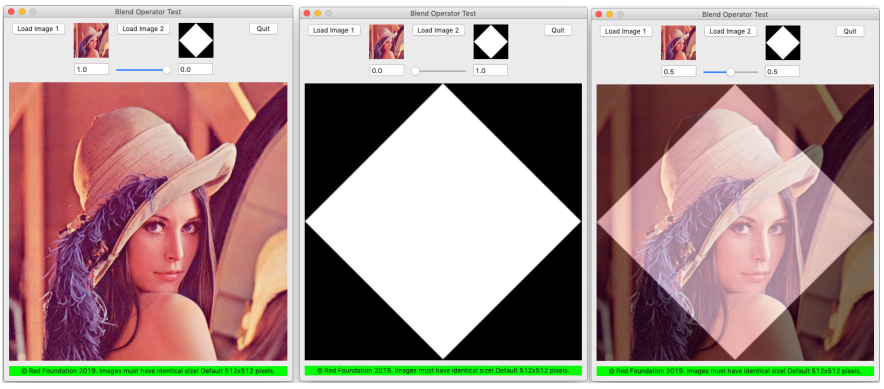
fading.red: calls *rcvPow*. Now if we want create a fading effect, playing with a value may be not sufficient. With *rcvPow* function (destination image: source image ^ value) a float value between 1.0 and 0.0 is used to modify all argb image values. This makes a nice rendering, like in movies, allowing the picture to progressively appear or disappear according to slider value.



In image processing we often use an operator with a first image and a second operator with another image. Sometimes, it's very interesting to mix the result of both operators. Blending functions present in redCV can help you for this purpose. **Attention: both images must have identical size!**

blendImages1.red: calls *rcvBlend*. The idea here is to combine alpha value of the second image as complement (1.0 – alpha) of the alpha value of the first image.

Alpha image1: 1.0	Alpha image1: 0.0	Alpha image1: 0.5
Alpha image2: 0.0	Alpha image2: 1.0	Alpha image2: 0.5
Image 1 visible	Image2 visible	Both images are mixed

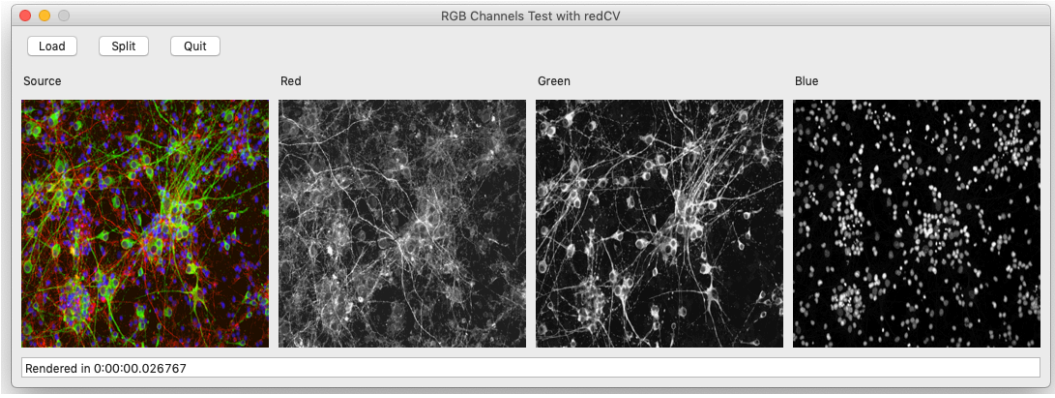


blendImages2.red: calls *rcvSetIntensity*. Similar to blendImages1.red, but uses *rcvSetIntensity* and not *rcvBlend* function.

blendMatrices.red: calls *rcvBlendMat*. For matrices fading.

redCV/samples/image_channel

These functions are basic operations we can use with redCV. Sometimes, it's very important to split and merge image channels for a better detection of image details. For example, we do often use colorized-cells images for identifying or counting the number of biological cells.



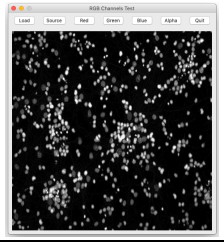
As can be seen in left original image, counting the number of neurons is difficult due to the density of connecting fibers between neurons. But splitting this image allows us (blue channel: right image) to better identify cells present in image.

redCVChannels. In this code we do not use routines but only red language. This works but is rather slow: **1958.328 ms** for processing the neurons image.
redChannels.red: calls *rcvSplit* function which splits each image channel. Red/System routines called by *rcvSplit* are used to improve speed processing: **27.176 ms** for the same image!
All job is done by splitImage function:

```
splitImage: function[][
        if isFile [
                t1: now/time/precise
                rcvSplit/red rimg imgR
                rcvSplit/green rimg imgG
                rcvSplit/blue rimg imgB
                canvasR/image: imgR
                canvasG/image: imgG
                canvasB/image: imgB
                sb1/text: copy "Rendered in "
                append sb1/text form now/time/precise - t1
        ]
]
```

redCVSplitMerge.red is similar but uses *rcvMerge* to create destination image from the 3 source images.

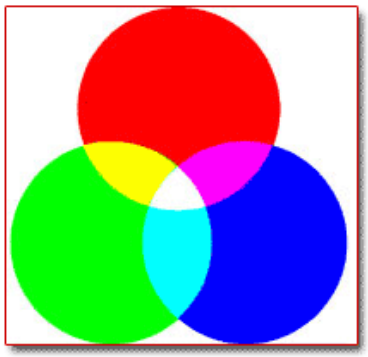
splitChannels: also calls *rcvSplit* function. Illustrative application (blue channel selected).



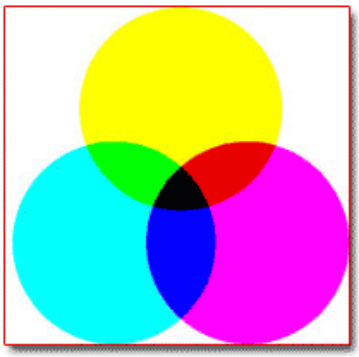
redCV/samples/image_colors

Red language perfectly controls additive and subtractive color synthesis. For example, you can type in red console *"red + green"* and you'll get *"255.255.0"* which corresponds to yellow color.

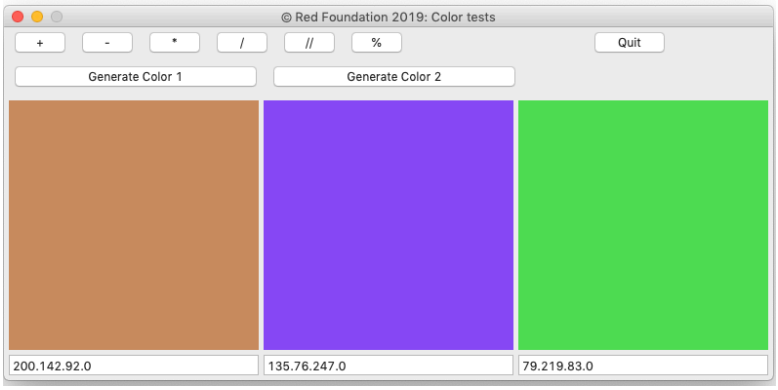
Additive



Subtractive



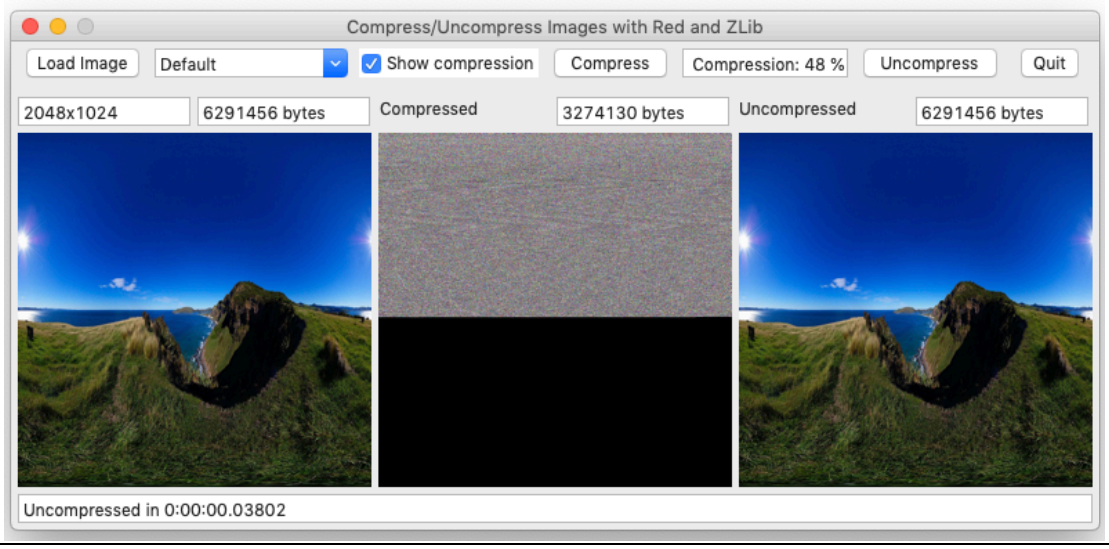
AdditiveColor.red and **SubtractiveColor.red**: both call *rcvColorImage* function to set both images to the required color and then call *rcvAdd* for combining the result in result image.
testcolor.red uses several redCV functions: *cvRandomImage/uniform* to create images with random color, *rcvGetPixel* to get back the color of generated image and lastly different mathematical operators on image (*rcvAdd*, *rcvSub*, *rcvMul*, *rcvDiv*, *rcvMod* and *rcvRem*).



redCV/samples/image_compression

ZLib is the sole external dependency library used by redCV. For macOS and Linux users, ZLib is generally installed. For Windows users, you'll find in /libs/ZLib/dlls the zlib1.dll file. Just copy the dll to your computer (e.g. c:\red\zlib1.dll) and all will be fine. ZLib access is based on Bruno Anselme's ZLib binding for Red language. You can use 3 levels for compression: cprx: ["Best speed" "Best compression" "Default"]

compress.red: A part of classical functions for loading and creating images, the code calls different functions and routines:
rcvLoadImageAsBinary: this function allows to directly load image/rgb values as a string of binary values that is mandatory for compression and decompression.
rcvCompressRGB: returns compressed binary data according to compression level (integer value [1: best speed, 9: best compression or -1: default compression]
rcvDecompressRGB: returns compressed data as uncompressed binary values. In order to get a perfect uncompressing process, **the size of original rgb data is required**.

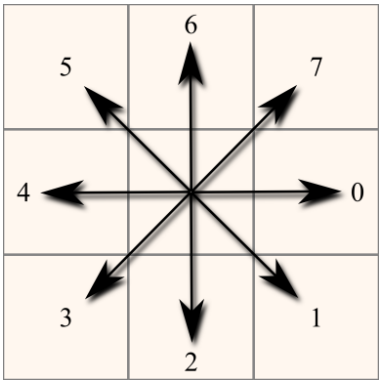


redCV/samples/image_contours

Finding contours of objects in image is a classical problem in image processing and different solutions are possible with redCV. A first solution is to use **Freeman Code Chain coding**.

Freeman Code Chain

A chain code is a way to represent shapes in a binary image. The basic idea is very simple: One spot on the outer boundary of a shape is selected as the starting point (generally left/up first point), we then move along the boundary of the shape and describe each movement with a directional code that describes the movement. We continue tracing the boundary of the shape until we return to the starting point.



The Freeman code encodes the movement as an integer number between 0 and 7.

0: east
1: southeast
2: south
3: southwest
4: west
5: northwest
6: north
7: northeast

The idea is to look for the neighbors of the current pixel and get the direction according to the neighbor value. We do implement Freeman code chain in redCV, but this requires some image preprocessing. First of all, we need a binary image. This is easily done with 2 redCV functions: *rcvImage2Mat* which transforms any red image to a binary matrix [0..255] and *rcvMakeBinaryMat* which makes a [0..1] matrix.

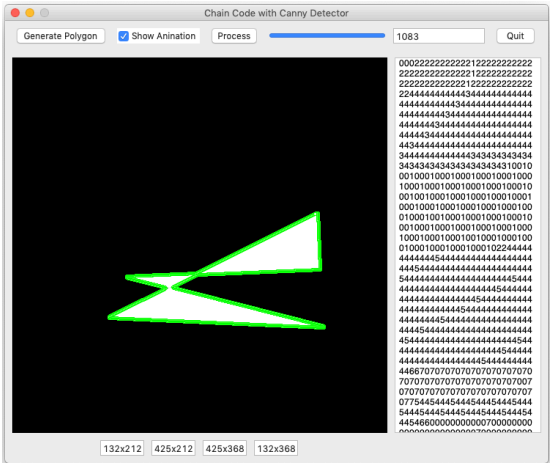
Then we have to get all points that belong to the shape. Just call *rcvMatGetBorder* function that looks for all value (0 or 1) that are part of shape boundary. Found pixels in binary matrix are stored in a block such as border. When found you have to copy pixels from border to another matrix which will be used to store visited pixel value.

Then redCV uses the *rcvMatGetChainCode* to get the successive movement direction such as in the code sample:

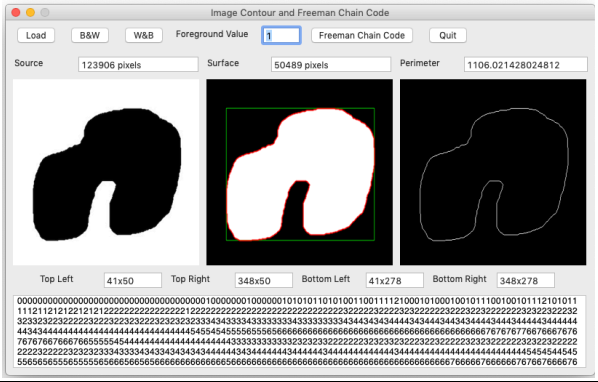
```
perim: length? border
p: first border
i: 1
while [i < perim] [
        d: rcvMatGetChainCode visited iSize p fgVal
        idx: (p/y * iSize/x + p/x) + 1
        visited/:idx: 0; pixel is visited
        append s form d
        switch d [
                0       [p/x: p/x + 1]              ; east
                1       [p/x: p/x + 1 p/y: p/y + 1] ; southeast
                2       [p/y: p/y + 1]              ; south
                3       [p/x: p/x - 1 p/y: p/y + 1] ; southwest
                4       [p/x: p/x - 1]              ; west
                5       [p/x: p/x - 1 p/y: p/y - 1] ; northwest
                6       [p/y: p/y - 1]              ; north
                7       [p/x: p/x + 1 p/y: p/y - 1] ; northeast
        ]
        i: i + 1
]
```

It is very important to ensure that the current pixel was processed; if not, the pixel can be processed again and the chain code could be invalid. This is why we set to 0 visited pixel in shape matrix (visited/:idx: 0; pixel is visited). Then, according to the direction value returned by *rcvMatGetChainCode* function we move to the new pixel to be analyzed.

freeman.red: basic application which generates rectangles and returns Freeman code chain.
freemanregular.red: this application generates regular shapes (square, circle, triangle and regular polygon), returns Freeman code chain and shows detected contours.
freemanirregular.red: a more sophisticated application for **irregular polygons** that can be difficult to detect. This program used supplementary functions to improve polygons detection. First, we use we **convolute** the image with a kernel [-1.0 -1.0 -1.0 -1.0 8.0 -1.0 -1.0 -1.0 -1.0] (*rcvConvolve*: see image_filters/ for convolution samples) in order to detect shape contours. However, edges filter can create discontinuities in detected contour. This why we use a **morphological operator** (*rcvDilate*: see image_morphology/ for samples), which dilates shape and suppress 0 values potentially created by filter. As illustrated by the next figure, result is pretty good.



imagecontour.red: if you preprocess your image with *rcv2WB* or *rcv2BW* functions in order to get continuous binary images, Freeman code chain can be used for any shape as illustrated by the result of this sample.



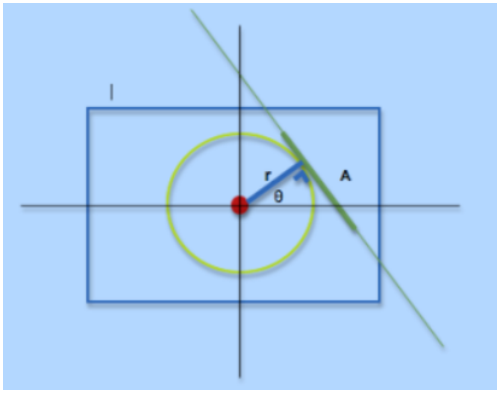
In this code, first left, first up, first right and first bottom pixels are first identified in the matrix in order to detect the position of the shape across the image (green rectangle).

lPix: rcvMatleftPixel bmat matSize fgVal
rPix: rcvMatRightPixel bmat matSize fgVal
uPix: rcvMatUpPixel bmat matSize fgVal
dPix: rcvMatDownPixel bmat matSize fgVal

Then, border pixels are identified with *rcvMatGetChainCode* and only external pixels (in red) are selected to create a new image with only detected contour.

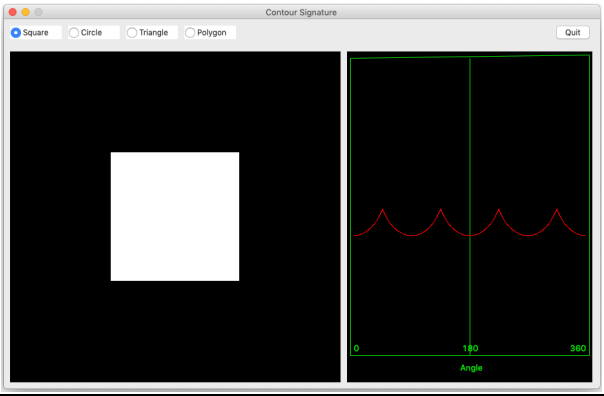
Shape signature

Now, an interesting way for processing contour in image is to look for **shape signature**. This can be easily done by combining Freeman code chain and **polar coordinates transform**. Basically, each pixel that belongs to a shape can be defined, in polar coordinate system, by 2 values: *rho* which is the distance of the pixel to the centroid of the shape and *theta* which is the angle from the horizontal x axis.



© Bruno Keymolen

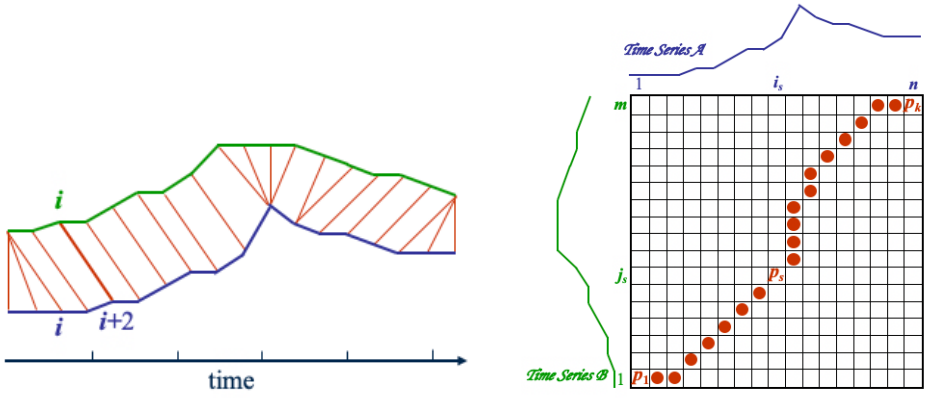
signaturePolar.red: By using *rcvMatGetChainCode* function to identify external pixels of the shape and the polar coordinates transform functions (*rcvGetEuclidianDistance* for rho and *rcvGetAngle* for theta), it's very easy to get the shape signature as a 1-D series of coordinates which varies in x-axis from 0 to 359 degrees.



You can also use *rcvGetAngleRadian* function which normalizes x values in [-pi +pi] space. Y-axis normalization [0.0..1.0] is also useful. This is done by *rcvRhoNormalization* function which finds the maximal rho distance (maxRho) in the distribution of rho values and then replaces each rho value by normalized rho equal to rho value * (1.0 / maxRho).
signatureCosine.red is similar but uses cosine law for calculating theta angles.

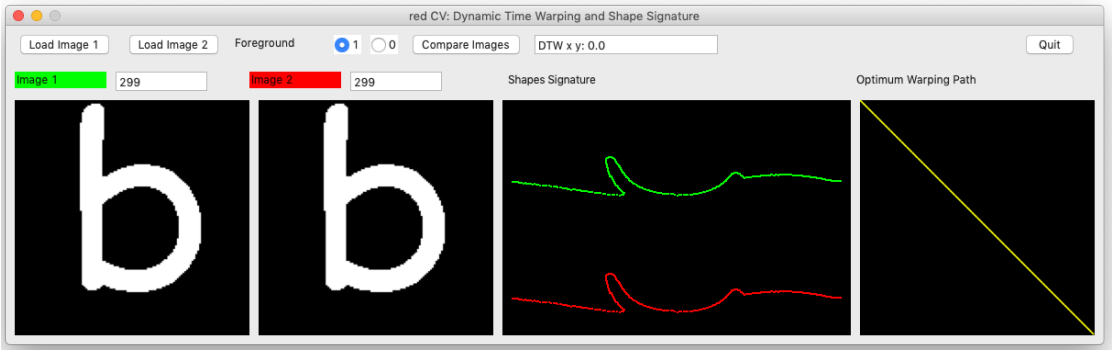
Dynamic Time Warping

Since we are able to identify shape signature as a 1-D series, it is possible to directly compare different shapes to look for the *similarity between shapes*. This is why redCV includes **Dynamic Time Warping (DTW) algorithm**. DTW is presented in detail in /TimesSeries/. For the moment, just consider that DTW is a non-linear alignment that produces an intuitive similarity measure, allowing similar shapes to match even if they are out of phase in the x axis such as time or space.



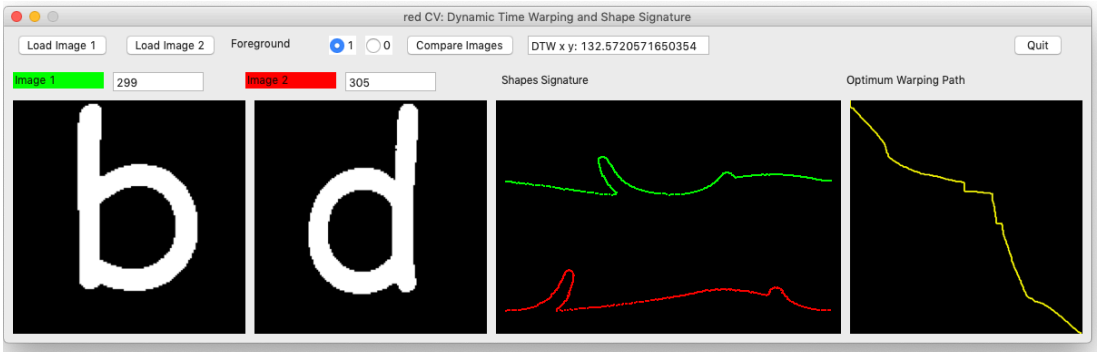
© Elena Tsiporkova

To find the *best alignment* between ***2 series*** A and B we need to find the path through the grid which *minimizes* the total distance between series: this is a *p warping function*.
Now imagine we have to compare characters to measure similarity between shapes such as hand-writing productions for example: DTW gives us a direct measurement of the distance between characters.
If the characters are identical, DTW equals to 0 as illustrated here:

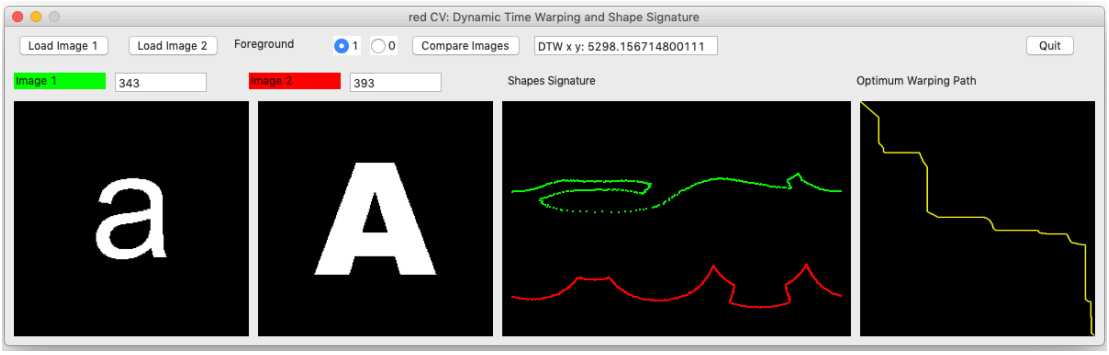


Shapes are identical and, of course, shape signatures are also identical. In this case the optimum path is the diagonal of the grid.

Now consider b and d characters that are close but different by orientation. In this case, DTW value increases.



If we compare now a and A, DTW is very high and shows that both shapes are not similar.



dtwFreeman.red and **dtwPolar.red** illustrate this technique for comparing shapes in image. dtwFreeman.red is a fast version that use only Freeman code chain to identify external pixels of shapes to be compared. **dtwPolar.red** is more complete since the code associates Freeman code chain and polar coordinates transform to creates X and Y DTW series. Both programs use *rcvDTW* functions: *rcvDTWDistances*, *rcvDTWCosts* and *rcvDTWGetDTW.*

redCV/samples/image_conversion

Code in this directory illustrates how redCV can be used for **color space transformation**. A color space is a method by which we can specify, create or visualize color. As humans, we may define a color by its attributes of brightness, hue and colorfulness. A computer may describe a color using the amounts of red, green and blue required to match a color. A printing system may produce a specific color in terms of the reflectance and absorbance of cyan, magenta, yellow and black inks on the printing paper. A color is usually specified using three parameters. These parameters describe the position of the color within the color space being used. Different space colors can be used with redCV:

RGB (Red Green Blue)
This is an additive color and basic system based on tri-chromatic theory. Used by systems, such as computers, that use a CRT to display images. RGB is easy to implement but non–linear with visual perception.

HSL (Hue Saturation and Lightness) and HSV (Hue Saturation and Value)-
This represents a set of similar color spaces. Most of these color spaces are linear transforms from RGB.

YCbCr (Luminance - Chrominance)
This is the television transmission color space. YCbCr is a digital standard. This color space separate RGB into luminance and chrominance information and is useful in compression applications for example.

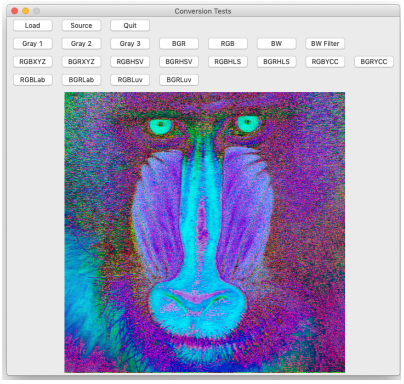
CIE
There are two CIE based color spaces, CIELuv and CIELab. They are nearly linear with visual perception.

Here is the list of between space colors conversions supported by redCV:

rcvInvert: Destination image: inverted source image
rcv2NzRGB: Normalizes the RGB values of an image
rcv2BW: Convert RGB image to Black[0] and White [255]
rcv2WB: Convert RGB image to White[255] and Black [0]
rcv2Gray: Convert RGB image to Grayscale
rcv2BGRA: Converts RGBA to BGRA
rcv2RGBA: Converts BGRA to RGBA
rcvIRgBy: Log-opponent conversion
rcvRGB2HSV: RBG color to HSV conversion
rcvBGR2HSV: BGR color to HSV conversion
rcvRGB2HLS: RBG color to HLS conversion
rcvBGR2HLS: RBG color to HLS conversion
rcvRGB2YCrCb: RBG color to YCrCb conversion

rcvBGR2YCrCb: BGR color to YCrCb conversion
rcvRGB2XYZ: RGB to CIE XYZ color conversion
rcvBGR2XYZ: BGR to CIE XYZ color conversion
rcvRGB2Lab: RBG color to CIE L*a*b conversion
rcvRGB2Lab: RBG color to CIE L*a*b conversion
rcvRGB2Luv: RBG color to CIE L*u*v conversion
rcvRGB2Luv: RBG color to CIE L*u*v conversion

conversion.red can be used for testing color space transformations: here RGB to HSV



redCV/samples/image_convolution

Convolution operator is extensively used in image processing to reduce the noises and find details in digital images. A 2D convolution can be thought of as replacing each pixel with the weighted sum of its neighbors. The kernel is another matrix, usually of smaller size, which contains the weights.

$$\begin{pmatrix} A_{-1,-1} & A_{0,-1} & A_{1,-1} \\ A_{-1,0} & A_{0,0} & A_{1,0} \\ A_{-1,1} & A_{0,1} & A_{1,1} \end{pmatrix}$$

A 3x3 convolution kernel

$$\begin{pmatrix} P_{x-1,y-1} & P_{x,y-1} & P_{x+1,y-1} \\ P_{x-1,y} & P_{x,y} & P_{x+1,0} \\ P_{x-1,y+1} & P_{x,y+1} & P_{x+1,y+1} \end{pmatrix}$$

Pixel neighbors

$$\begin{pmatrix} A_{-1,-1} * P_{x-1,y-1} & A_{0,-1} * P_{x,y-1} & A_{1,-1} * P_{x+1,y-1} \\ A_{-1,0} * P_{x-1,y} & A_{0,0} * P_{x,y} & A_{1,0} * P_{x+1,0} \\ A_{-1,1} * P_{x-1,y+1} & A_{0,1} * P_{x,y+1} & A_{1,1} * P_{x+1,y+1} \end{pmatrix}$$

Product of the kernel by the neighbors

Each product is the color value of the current pixel or a neighbor, with the corresponding value of the filter matrix. Each pixel value of the image is replaced by the sum of the product. Depending on the used filter, the resulting pixel values after applying the filter can be negative or larger than 255. You can truncate resulting values so that values smaller than 0 are set to 0 and values larger than 255 are set to 255. For negative values, you can also take the absolute value. Generally, the sum of the weights of the kernel is equal to 0 or 1. If this is not the case, the sum of weights can be computed during the computation, and used to scale the result. In this case, the best way is to set the factor parameter to 1/sum of weights to make the normalization. You can also call *rcvFilter2D* which gives the same result, since the sum of weights is computed during the summation, and used to scale the result.

Many filters (see /image_filters) are based on 2-D convolution. The 2-D convolution operation isn't extremely fast, unless you use small kernels. There are a few rules about the kernel. Its size has to be generally uneven, so that it has a center, for example 3x3, 5x5, 7x7 or 9x9 kernels are perfect. Apart from using a kernel matrix, convolution operation also has a multiplier factor and a delta factor. After applying the kernel, the factor will be multiplied with the result, and the bias added to it. So, if you have a kernel with an element 0.25 in it, but the factor is set to 2, all elements of the kernel are multiplied by two so that element 0.25 is actually 0.5.

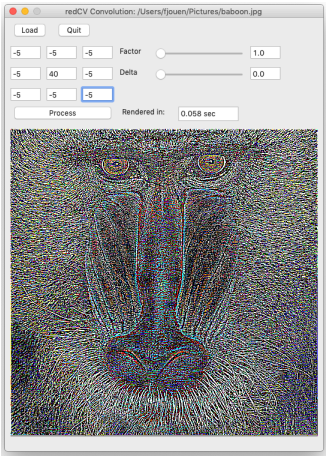
The multiplier factor is useful when the sum of weights is greater than 1. The delta factor can be used if you want to make the resulting image brighter or lighter.

convolution1.red and convolution2.red illustrate redCV convolution function. First, we have to define a kernel. Basically, Kernel is a NxM array and should be written as knl: [[-1.0 -1.0 0.0] [-1.0 0.0 1.0] [0.0 1.0 1.0]]. We just use a simple block! such as emboss3: [-1.0 -1.0 0.0 -1.0 0.0 1.0 0.0 1.0 1.0] for a better and quicker access to Red/System code required for convolution computation. So, kernel is just a block containing the values for the filter. Then we set factor to 1.0 and delta to 128.0 (in convolution1.red), and we call *rcvConvolve* image_source image_dst emboss3 factor delta. *rcvConvole* function reads all pixels of the image source, applies the filter according to factor and delta values and then returns the modified pixels in the destination image. **Convolution2** allows to play with factor and delta parameters.

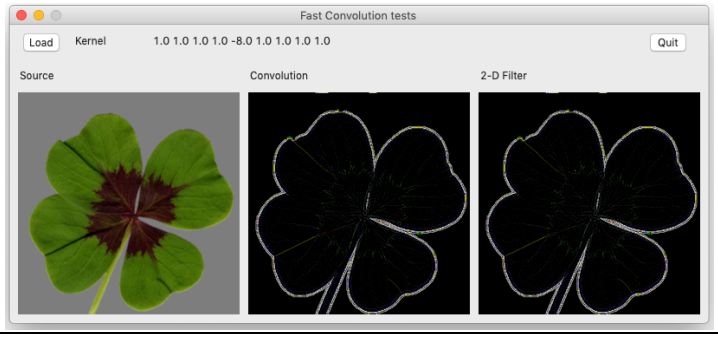
convolution1.red

convolution2.red

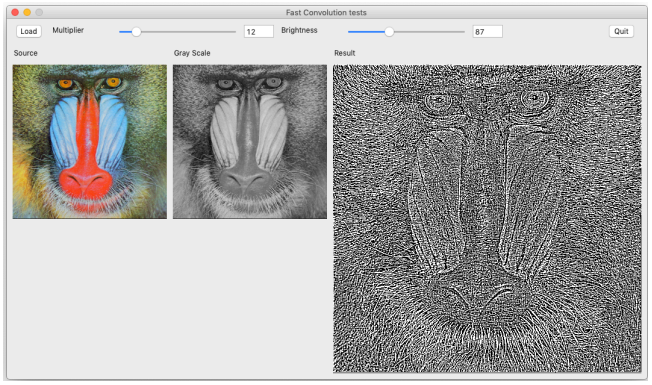
convolution3: also uses *rcvConvolve*. This code is for experiments. We can directly test the modification of the kernel values on the image.



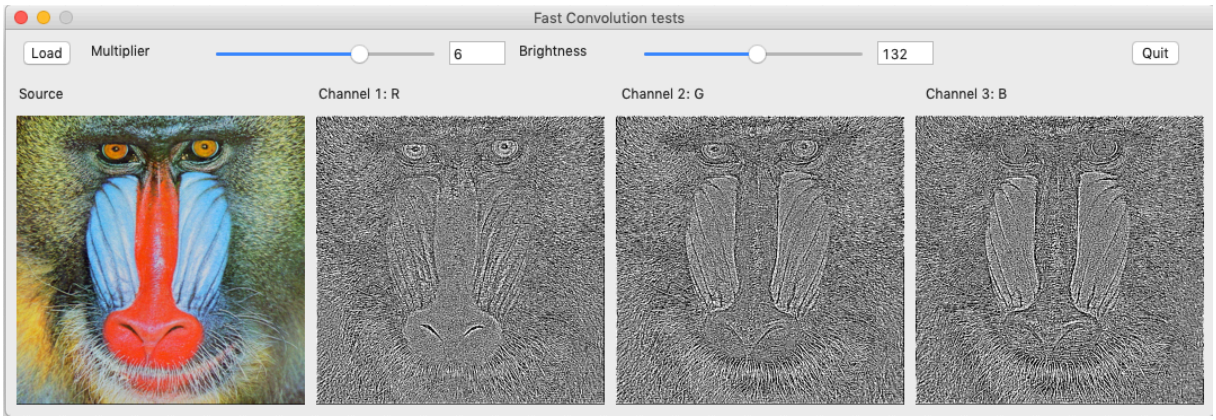
convolution4.red and convolution5.red: Both call *rcvFilter2D* which is similar to *rcvConvole* except that the sum of weights is calculated during the convolution and used to scale resulting value by diving the value by the sum of weights. This function allows to test filters which are not symmetrical and is very efficient pour Gaussian filter. When filters are symmetrical both functions return the same image as illustrated by convolution5.red.



fastConvolution1 and fastConvolution2. As previously underlined, convolution is not very fast. This is why we include in redCV a fast convolution function *rcvFastConvolve* which works on **one channel of the image for faster calculation**. The basic idea is to transform the source image into a gray scale image (**fastConvolution1**) or to process each RGB channel of the image (**fastConvolution2**).



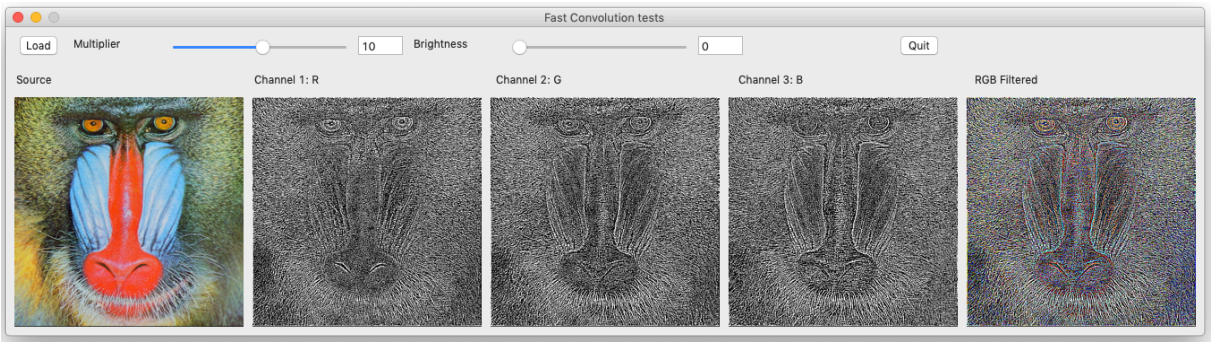
fastConvolution1



fastConvolution2

Since each channel is a 2-D matrix of red/System byte! datatype, processing is very fast and then you can merge processed channels into a new image.

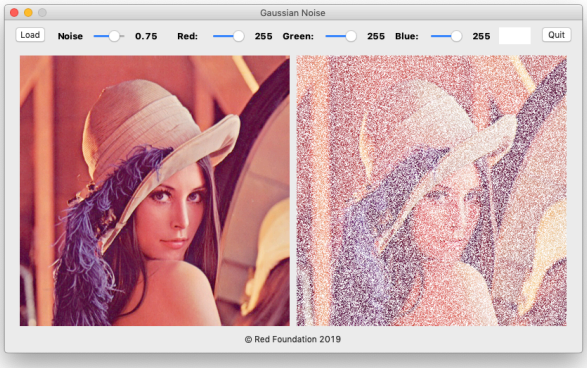
fastConvolution3: calls *rcvSplit2Mat*, *rcvConvolveMat*, *rcvMat2Image*, and *rcvMerge2Image*. In redCV, convolution can be also used with matrices for a very fast image processing. In this sample, image source is first separated into 4 matrices corresponding to image channels. Then, convolution is performed on each rgb matrix and the result is displayed with *rcvMat2Image* function for each image channel. Lastly, all matrices are merged into destination image which displays the result of the filtering with the *rcvMerge2Image* function.



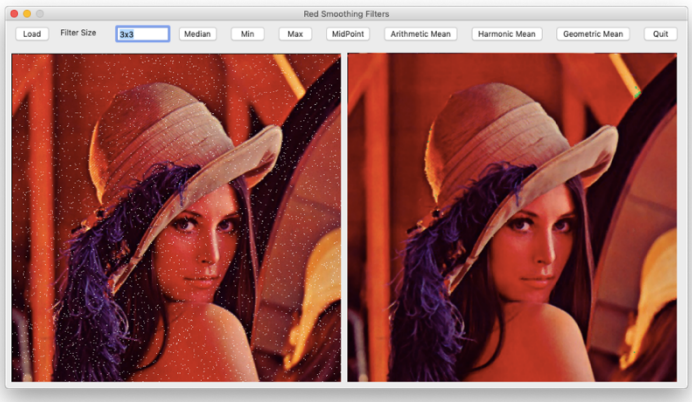
redCV/samples/image_denoising

redCV can be used for **image denoising**. A lot of functions are included for helping image restoration. Basically, a 3x3 kernel is used to calculate the pixel value of neighbors and to replace the pixel value in the image by the result of different algorithms. Of course, kernel size can be changed. According to the noise included in image you can use different parametric filters.

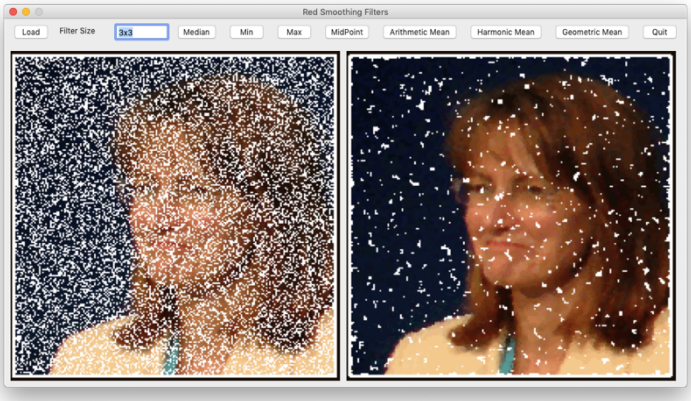
randomNoise.red. First of all, redCV includes *rcvImageNoise* function which allows to add a Gaussian noise on any image. Since we are using RGB images, noise can be added by combining each RGB value. Noise is a float value [0.0..1.0] which determines the number of pixels to be randomly replaced by color.



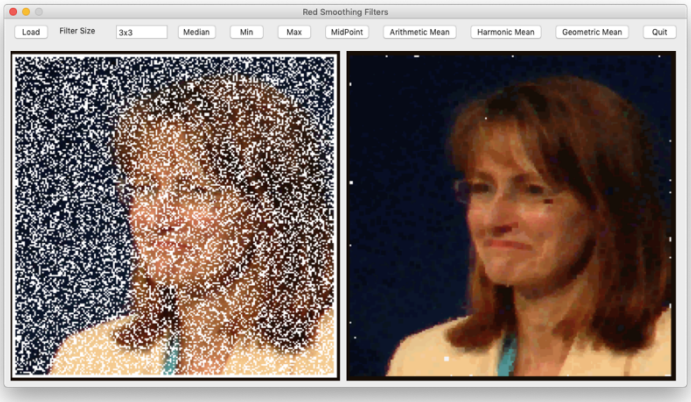
When noise is simple, such as *pepper and salt noise,* a simple median filter will be efficient: Central pixel value is replaced by the median value of neighbors by *rcvMedianFilter* function. You can also call **rcvMeanShift** function for denoising images (see redCV/samples/image_histograms/meanShift.red)



But when the image is really noisy, with a *Gaussian noise* for example, median filter is not sufficiently efficient:

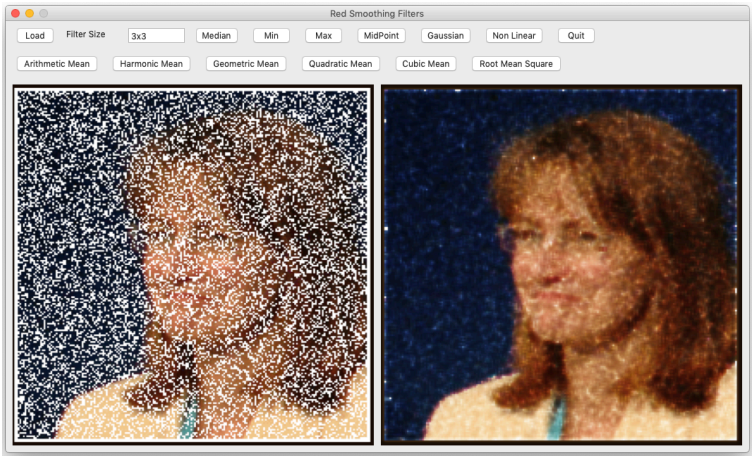


In this case, *rcvMinFilter* function can be used: Central pixel value is replaced by the minimum value of neighbors and the result is pretty good.



You can also use *rcvMaxFilter* (max value of neighbors) or *rcvMidPointFilter* (central pixel value is replaced by minimum+ maximum values of neighbors divided by 2) according to the noise contained in the image. Gaussian filter (*rcvGaussianFilter*) and nonlinear filter (*rcvNLFilter*) are also present in redCV smoothing functions.

smoothing.red: redCV also includes a *rcvMeanFilter* function, which can be used for image smoothing.



rcvMeanFilter supports different mean calculations that are made **for each image channel**:
arithmetic mean: 1/n * (x1+ x2 + ….xn)
harmonic mean: n / (1/x1 + 1/x2 + … 1/xn)
geometric mean: power (x1 * x2 * … xn) 1/n
quadratic mean: sqrt (x1*x1 + x2 * x2 + … xn * xn / n)
cubic mean: power (x1*x1 + x2 * x2 + … xn * xn / n) (1.0 / 3.0)
root mean square: sqrt (1/n * (x1*x1 + x2 * x2 + … xn * xn))

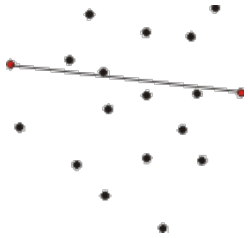
redCV/samples/image_detectors

In this section you'll find some classical algorithms that can help you to detect objects in images.

Convex Hull

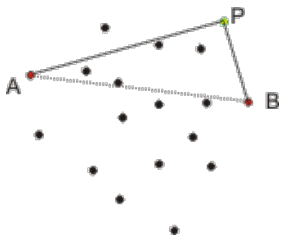
The convex Hull problem in geometry tries to find the smallest convex set containing the points present on the image. There are many approaches for handling this problem, but for redCV we focused on the *Quick Hull algorithm*, which is one of the easiest to implement and has a reasonable expected running time of *O(n log n)*. A clear explanation of the algorithm can be found here: http://www.ahristov.com/tutorial/geometry-games/convex-hull.html. Thanks to Alexander Hristov for the original Java code. Since this website seems dead, I'll shortly present Alexander's approach.
Imagine a set of random points as illustrated below.

As a start for the algorithm, we need to find a segment that belongs to the final convex hull. One such segment is the line between the leftmost and the rightmost points, which are known to be inside the hull. This segment is used the initialize a set of hull points.

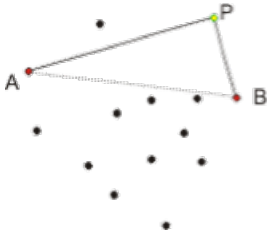


Now it is easy to divide the set of the remaining points into two sets, depending on which side of the line they fall. Let's call S1 the set of the points above the line, and S2 the set of the points below the line. For each of the sets, we do proceed **recursively** as follows:

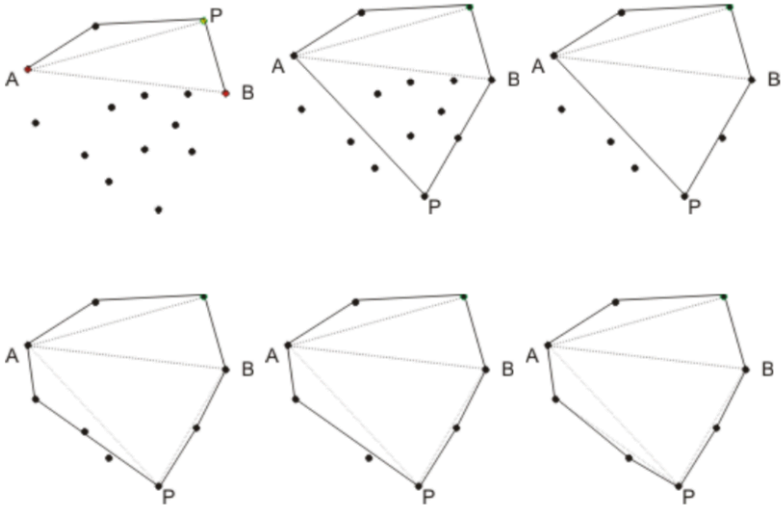
1°) We look for the point (P) that is furthest away from the line. This point belonging to the hull, is inserted in the set of hull points between A and B.



2°) We remove from S1 set all points that are inside the triangle.

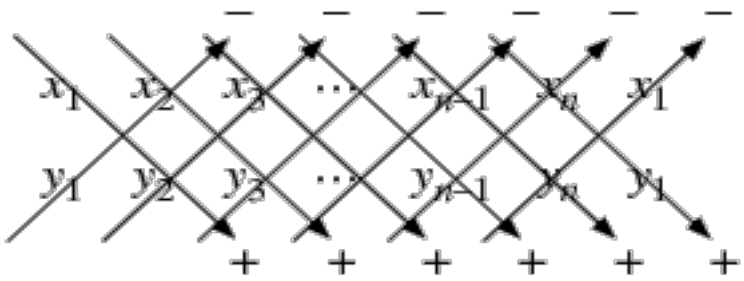


3°) Again, we calculate two sets: The set of points above AP and the set of points above PB (in the example above, this set is empty). We replace the original line AB with the segments AP and PB, and recursively apply the algorithm on the two sets. We stop when a set is empty or has only one point, in which case the point belongs to the hull. The whole recursive process is illustrated below.



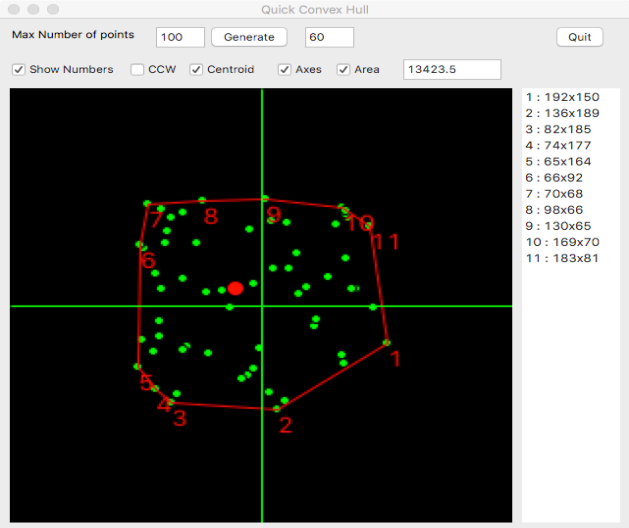
We do also develop *rcvContourArea* function, which calculates the area of polygon generated by rcvQuickHull function. This function returns the (signed or not) area (A) of a planar non-self-intersecting polygon with vertices (x_1,y_1), ..., (x_n,y_n) according to the formula:

$$A = \frac{1}{2}\left(x_1 y_2 - x_2 y_1 + x_2 y_3 - x_3 y_2 + \ldots + x_{n-1} y_n - x_n y_{n-1} + x_n y_1 - x_1 y_n\right),$$

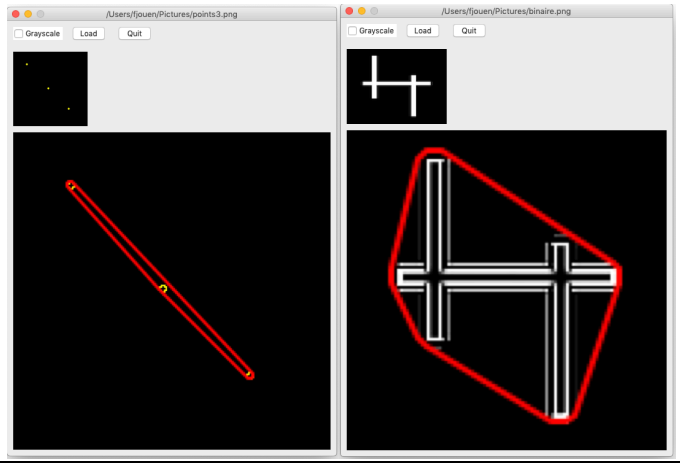


You'll find here, Weisstein, Eric W. "Polygon Area." From MathWorld--A Wolfram Web Resource. http://mathworld.wolfram.com/PolygonArea.html, a very clear demonstration of the algorithm.

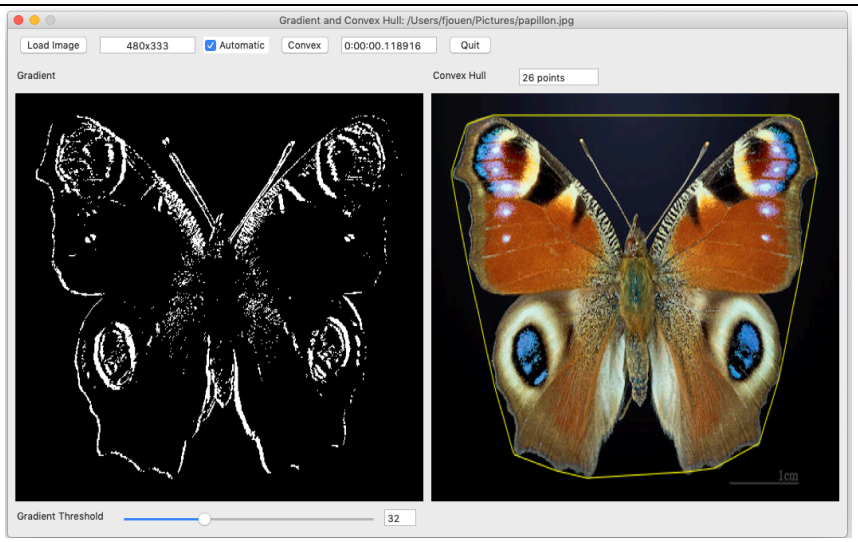
convexArea.red illustrates the use of *rcvQuickHull* and *rcvContourArea* functions.



pointDetector.red. In the previous sample, points are randomly generated and directly stored in a block used by *rcvQuickHull* function. Now the question is how to do when dots are in any image. *rcvPointDetector* function can help us. This function applies a Canny-like filter on image in order to enhance contour detection. In association with the *rcvGetPairs* function, that allows to return a block of coordinates of pixels of the shape, applying *rcvQuickHull* function is rather trivial and gives pretty good results.



gradient.red. Nevertleless, *rcvPointDetector* is very efficient with binary image and not really adapted for RGB image. As usual in image processing, we have to chain different operators to obtain the desired result. In this code, we illustrate how convex Hull algorithm can be associated to **gradient filtering** for processing any type of shapes in colored images. Here the idea is to use matrices in order to accelerate the whole process. Image is first loaded as a grayscale image in order to process only 1 channel (RGB channels are identical in grayscale) and copied to a matrix by the *rcvImage2Mat* function. Then, we use the *rcvMakeGradient* function which makes a gradient matrix for contour detection and returns max gradient value of the matrix. This function is similar to Sobel filter. *rcvMakeBinaryGradient* function transforms the previous matrix as a **binary matrix** [0..1] and keeps only original pixels greater than a threshold. This allows to eliminate some details outside of the shape. Then, the useful function *rcvGetPairs* is called to know coordinates of pixels. Once we get the coordinates, *rcvQuickHull* function can be called to do the job!

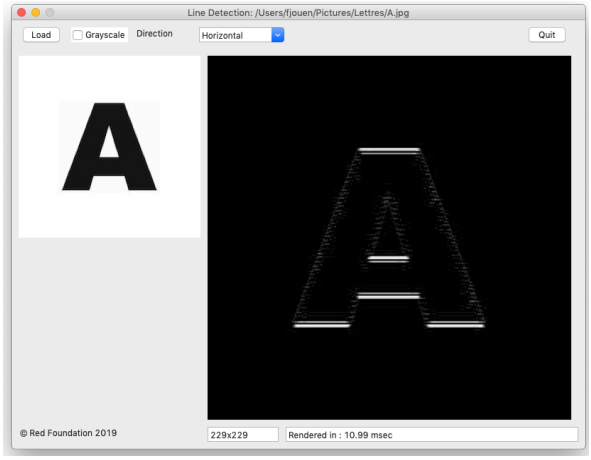


It is also possible to use other techniques, such as Freeman code chain algorithm, to make contour detection (see /images_contours/), but this one is rather fast due to the use of binary matrices.

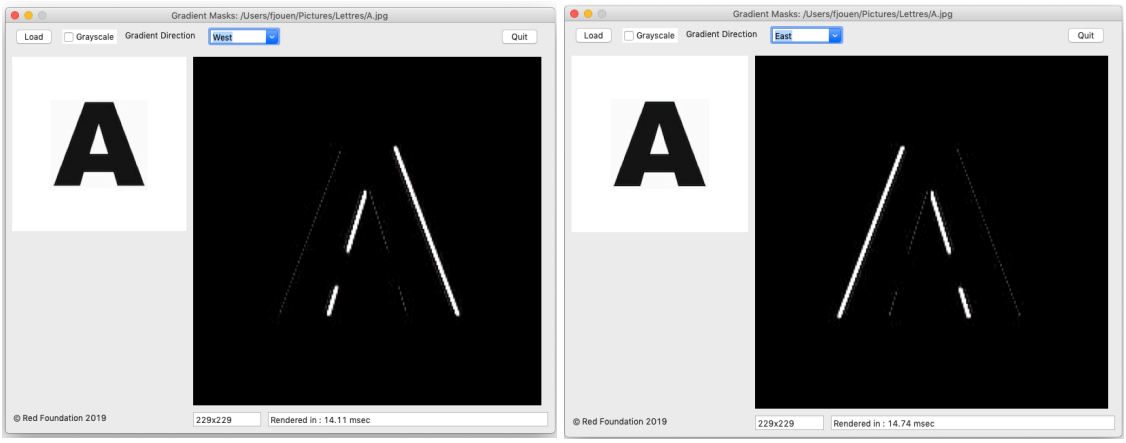
Line detection

Sometimes, we need to find lines inside a digital image. Some basic functions are included in redCV in order to solve this question. A very easy way is to use build-functions devoted to line detection.

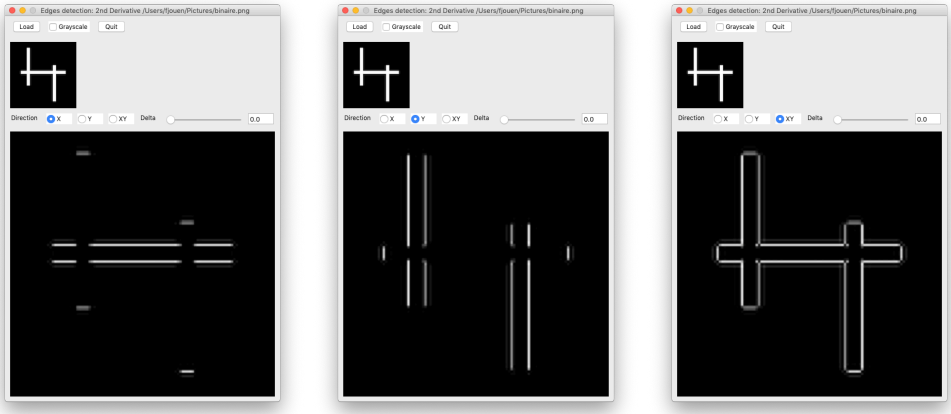
linedetector.red uses *rcvLineDetection* function which convolves the image with different kernels sensitive to horizontal, vertical, left diagonal and, right diagonal directions.



gradientsMask.red is similar but *rcvGradientMasks* function offers 8 directions and is very interesting for detection oblique lines in image.



derivative.red, which uses second derivative, is also performant for horizontal and vertical edges detection.

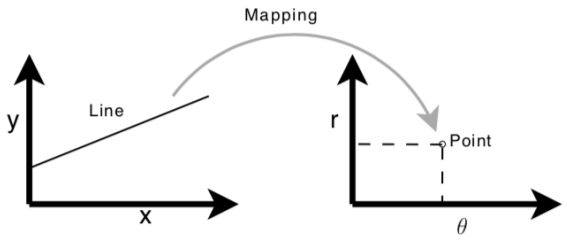

Edges detection: 2nd Derivative /Users/fjouen/Pictures/binaire.png
Load
Grayscale
Quit
Direction
X
Y
XY
Delta
0.0

Hough Transform

The Hough transform was originally developed to recognize lines (P.V.C. Hough. Method and means for recognizing complex patterns, U.S. patent 3069654. 1962.).
Lines can be represented by two parameters *a* and *b* according to the equation $y = a\,.\,x + b$. This equation is, however, not able to represent vertical lines and Hough Transform uses another equation which can be rewritten to be similar to the first equation.

$$r = x \cdot \cos\theta + y \cdot \sin\theta \Leftrightarrow$$
$$y = -\frac{\cos\theta}{\sin\theta} \cdot x + \frac{r}{\sin\theta}$$

The idea of the Hough Transformation is to represent lines in a polar coordinate system, in this case, called Hough Space. All lines can be represented in this form where θ ∈ [0,180[and r ∈ R (or θ ∈ [0,360[and r ≥ 0). The Hough space for lines has these two dimensions, θ and r, and a line is represented by a single point, corresponding to a unique set of parameters (θ0, r0). The line-to-point mapping is illustrated in the next figure.



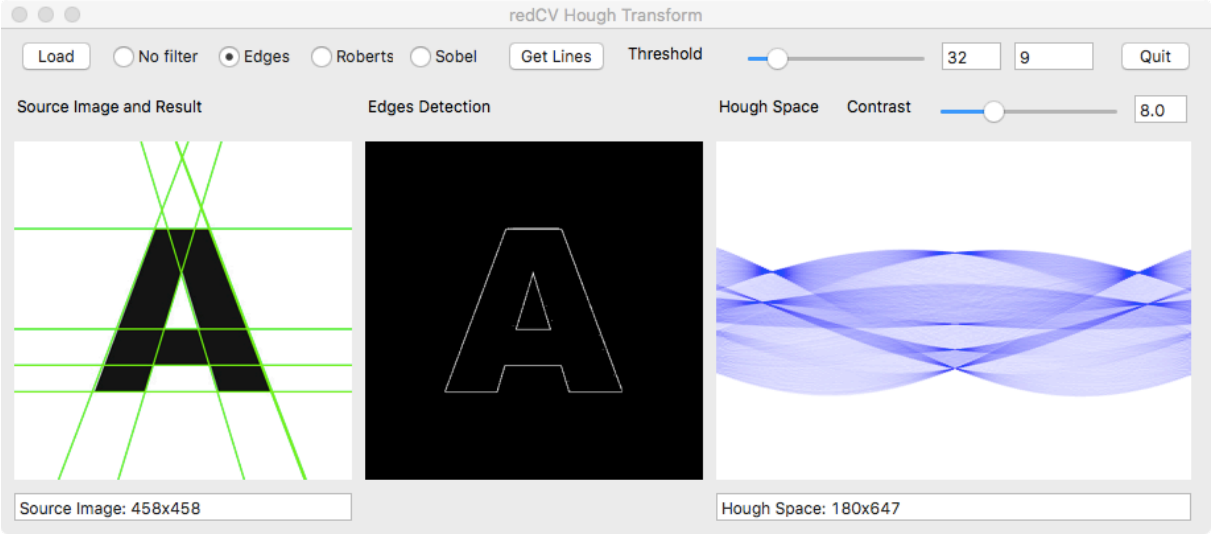
An important concept for the Hough transform is the mapping of single points. The idea is, that a point is mapped to all lines, that can pass through that point. This yields a sine-like line in the Hough space. Now, we need to add one extra element which is called an accumulator. An accumulator is a 2-dimensional matrix with θ max columns and r max rows. Each cell represents a unique coordinate (r, θ) and thus one unique line in Hough Space.

After, the algorithm is rather simple. First, we have to initialize the accumulator to 0. Then for every pixel in the image we compute r for θ = 1 to 180 (or 360) and increment the accumulator by 1 for every found r. When all pixels are processed, the accumulator contains a count of pixels for every possible line. Now it is possible to use a threshold (generally the size of the line) to keep lines that are over the threshold.
The result of the is a set of coordinates (r, θ), one for every straight line above the threshold. These polar coordinates can be easily computed back to 2 points in the original image space, just solving the equation for x and y:

$$r = x.cos(\theta) + y.sin(\theta)$$

$$x = \frac{r - y.sin(\theta)}{cos(\theta)} \qquad y = \frac{r - x.cos(\theta)}{sin(\theta)}$$

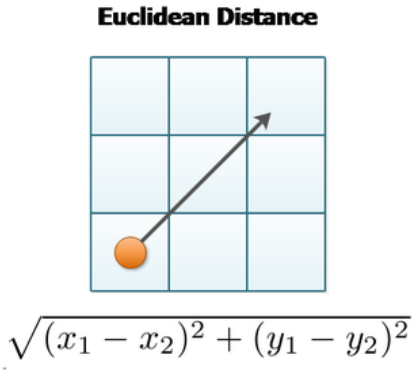
hough1.red and hough2.red illustrate the implementation in redCV. Hough1 only uses a Canny filter for edges detection. Hough2 includes a series of edges filters to creates a BW image with lines and points representing the edges. The program gives the number of detected lines, here 9, which are visualized in the original image.



redCV/samples/image_distances

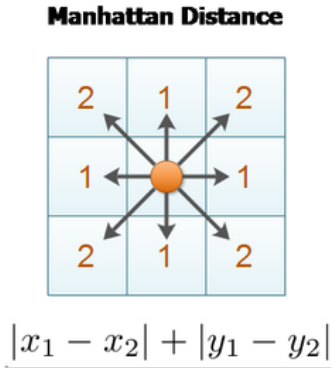
In image analysis, distance transforms are very important for many processing algorithms. Basically, in distance transform, binary image specifies the distance from each pixel to the nearest non-zero pixel. There are many metrics to calculate a distance between 2 points (x1, y1) and (x2, y2) in XY plane. redCV includes different functions for calculating distances between pixels.

rcvGetEuclidianDistance returns the straight-line distance between two points and is evaluated with the Euclidean norm.



https://lyfat.files.wordpress.com/2012/05/euclidean.png

rcvGetManhattanDistance measures the path between the points. The distance between two points is the sum of the absolute differences of their coordinates. You can also use *rcvGetChessboardDistance* function.

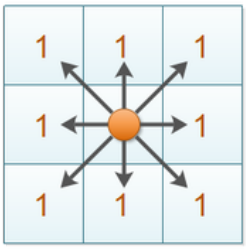


https://lyfat.files.wordpress.com/2012/05/manhattan.png

.

rcvGetChebyshevDistance is a generalization of the Minkowski distance for h (a real number) → ∞. To compute it, we calculate the maximum difference in values between the two points.

Chebyshev Distance



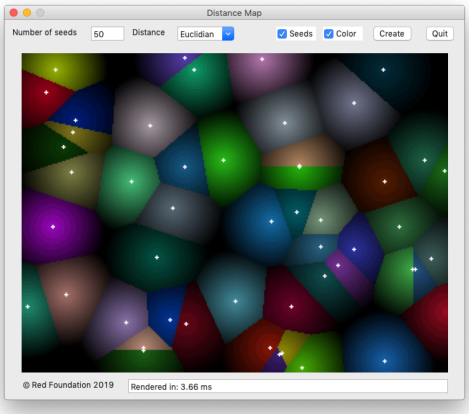
$$\max(|x_1 - x_2|, |y_1 - y_2|)$$

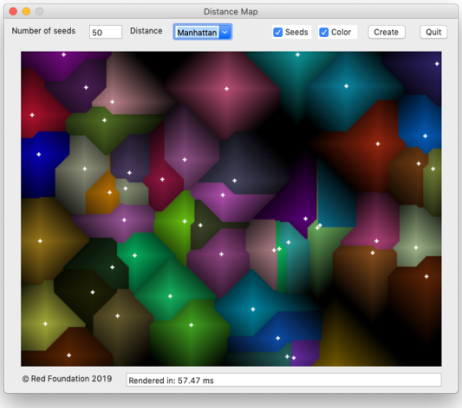
https://lyfat.files.wordpress.com/2012/05/manhattan.png

rcvGetMinkowskiDistance: The Minkowski distance can be considered as a generalization of both the Euclidian and the Manhattan distances. It represents the Manhattan distance when h = 1.0 and Euclidean distance when h = 2.0.

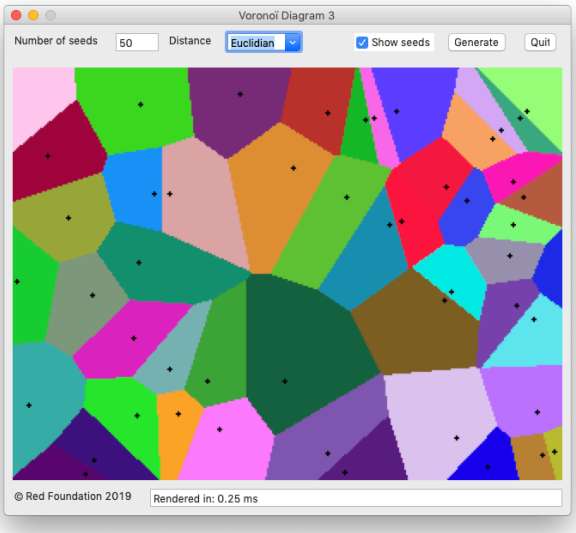
redCV also includes some fractional distances calculations such as *rcvGetCamberraDistance* and *rcvGetSorensenDistance* functions.

/voronoi/mapdiagram.red illustrates the effects of distance metric when generating a distance diagram. This code is based on Boleslav Březovsky's sample of distance mapping. It first creates some random points and then calculates distance from nearest peak from each point in image. The shorter the distance is, the brighter the pixel is.





/voronoi/voronoi.red is similar to the previous sample. The code shows the partitioning of a plane with n points into convex polygons such that each polygon contains exactly one generating point (a seed) and every point in a given polygon is closer to its generating point than to any other. A Voronoi diagram is sometimes called *Dirichlet tessellation*. The cells are called Dirichlet regions, Thiessen polytopes, or Voronoi polygons. Code gives also the opportunity to use different distance metrics such as Euclidian or Manhattan distances.



k-means clustering is a method of vector quantization, originally from signal processing, that is popular for cluster analysis in data mining. k-means clustering aims to partition n observations into k clusters in which each observation belongs to the cluster with the nearest mean, serving as a prototype of the cluster. This results in a partitioning of the data space into Voronoi cells (https://en.wikipedia.org/wiki/K-means_clustering).

You'll find in /kMeans different versions of k-means algorithm with Red and redCV. **kMeans.red** is 100% Red code which illustrates the implementation in Red language. Of course, this code is slow and is improved with Red/System routines (**kMeansRCV.red** and **kMeansRCV2.red**).

Attention: k-means code uses specific array (a block of vectors) for faster computation.

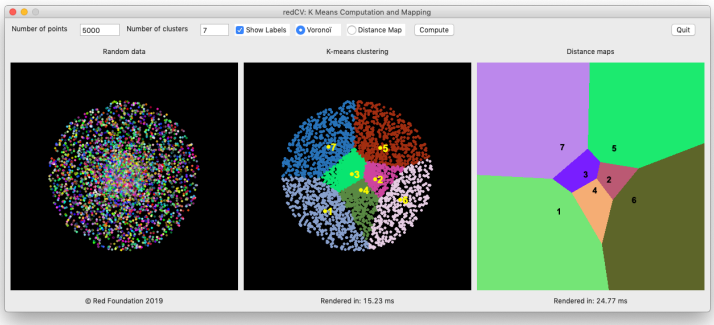
rcvKMInitData function allows to create this kind of arrays either for data or for centroid.

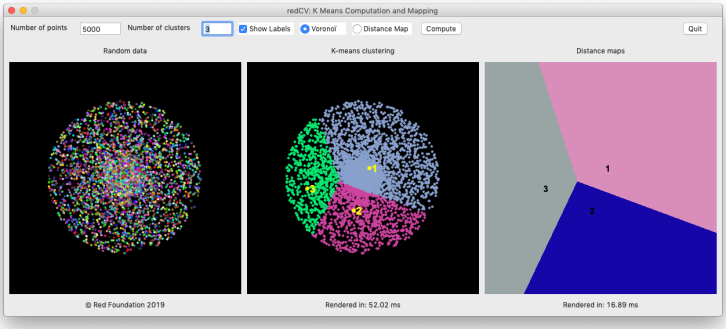
```
rcvKMInitData: function [count [integer!] return: [block!]]
"Creates data or centroid array"
][
	blk: copy []
	i: 0
	while [i < count] [
		append blk make vector! [float! 64 [0.0 0.0 0.0]]
		i: i + 1
	]
	blk
]
```

Exemple:
K: 7
centroid: rcvKMInitData K

rcvKMInit function uses **k-means++ algorithm** for choosing the initial values (or seeds) for the k-means clustering algorithm. This seeding method yields considerable improvement in the final error of k-means. Although the initial selection in the algorithm can take extra time, the k-means part itself converges very quickly after this seeding and thus the algorithm actually lowers the computation time.

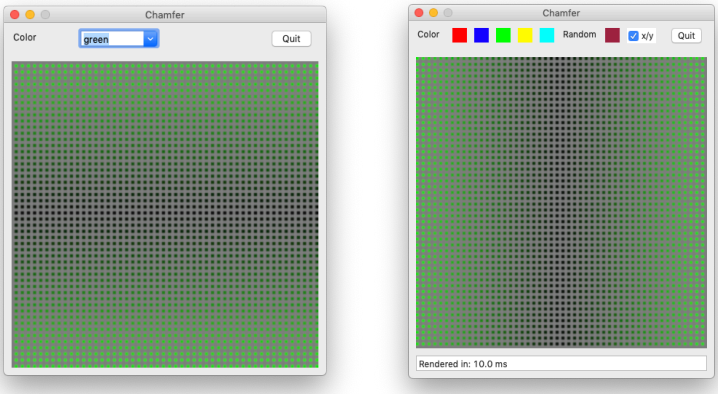
rcvKMCompute function: Given an initial set of k means, the *Lloyd K-means clustering* algorithm proceeds by alternating between two phases until convergence is reached. First, it assigns each observation to the cluster whose mean has the least squared Euclidian distance. Then, it calculates the new means (centroids) of the observations in the new clusters. The algorithm has converged when the assignments no longer change (when 99.9% of points are classified).



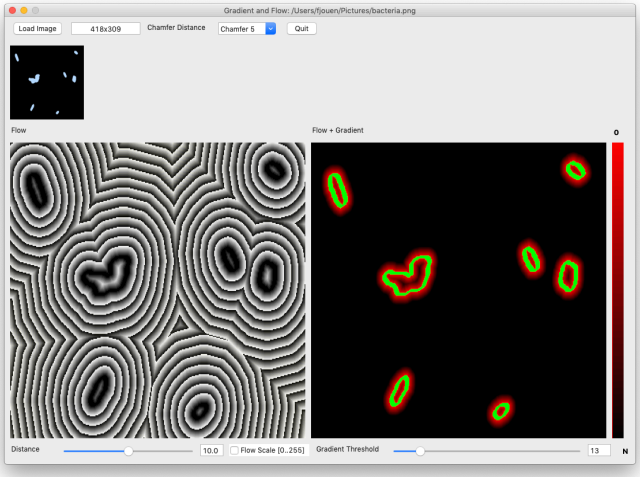


As previously underlined, distance transforms compute the distance of every pixel in a binary image to the nearest point in an object set (the object set may be a single point). However, distance transforms often use approximations to Euclidean distance for speed of processing generally using chamfer metrics. The basic idea here is to use predefined kernels with chamfer value to avoid root-square computation associated to Euclidian norm. With chamfer metrics, processing pixels distance only required a double-step image analysis: forward and backward scans from up-left to right-down pixel and from right-down to up-left pixel.

Chamfer/**chamfer.red** and **chamfer2.red** illustrates a simple implementation of chamfer distance with Red and Red/System routine. Of course, chamfer2.res is faster.



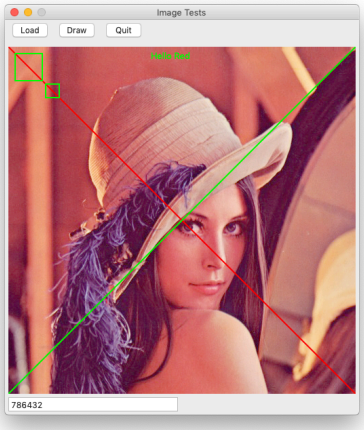
Chamfer/**flow.red** is 100% redCV implementation for fast and sophisticated applications including gradient and flow analysis. Basically, **flow.red** loads RGB images, makes a transformation to grayscale and then calls *rcvMakeBinaryGradient* function, which is similar to Sobel filter in order to identify objects in image. Then *rcvChamferCompute* function calculates Chamfer distance map then used by *rcvGradient&Flow* function to calculate gradient and flow values in image.



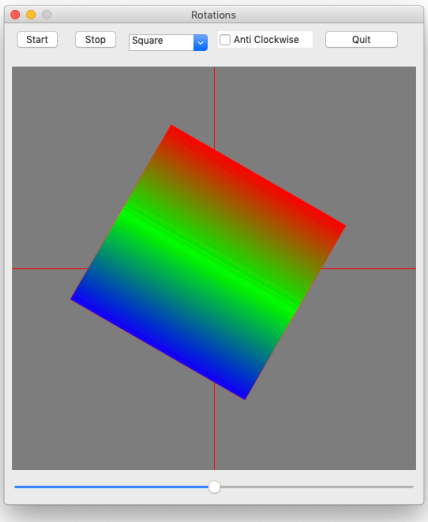
redCV/samples/image_draw

Code samples in this directory **do not use redCV**, but just illustrate how red draw dialect can be used with images. Many samples of redCV call draw for rendering. Other documentation can be found here: http://www.red-lang.org.

drawOnImage.red is really basic ☺

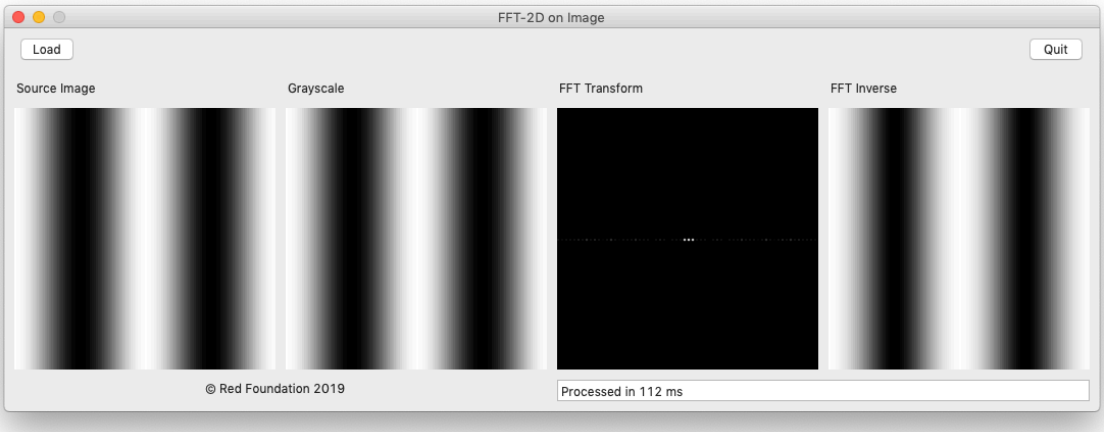


heaver.red and **rotations.red** are more complex since they are using gradients and timer to create nice animations.



redCV/samples/image_FFT

Thanks very much to Steven Lehar for his luminous explanation of the basis of FFT applied to image (http://cns-alumni.bu.edu/~slehar/fourier/fourier.html).
As underline by S. Lehar, Fourier theory states that any signal, including visual images, can be expressed as a sum of a series of sinusoids. For example, the following sinusoidal pattern can be captured in a single Fourier term that encodes the spatial frequency, the magnitude (positive or negative), and the phase of signal.



redCV/samples/image_filters

redCV includes a lot of predefined filters that use image convolution with a 3x3 kernel. Of course, if you want more sophisticated filters you can use the general *rcvConvole* function (see redCV/samples/image_convolution). The action of a convolution is simply to multiply the pixels in the neighborhood of each pixel in the image by a set of weights and then replace the pixel by the sum of the product. In order to prevent the overall brightness of the image from changing, the weights are either designed to sum to unity or the convolution is followed by a normalization operation, which divides the result by the sum of the weights. Thus, the filter is generated by providing a set of weights to apply to the corresponding pixels in a given size neighborhood. The set of weights make up what is called the convolution kernel and is typically represented in a table or matrix, where the position in the table or matrix corresponds to the appropriate pixel in the neighborhood. Such a convolution kernel is typically a square of odd dimensions so that, when applied, the resulting image does not shift a half pixel relative to the original image. The general form for a 3x3 convolution kernel looks as follows:

$$\begin{bmatrix} w1 & w2 & w3 \\ w4 & w5 & w6 \\ w7 & w8 & w8 \end{bmatrix}$$

or

$$\frac{1}{sumw}\begin{bmatrix} w1 & w2 & w3 \\ w4 & w5 & w6 \\ w7 & w8 & w8 \end{bmatrix}$$

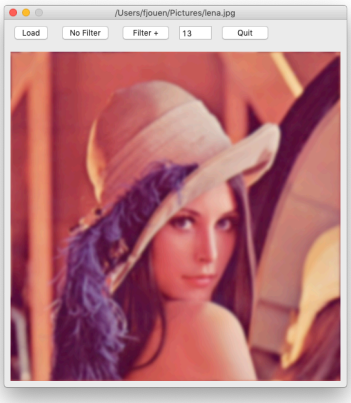
if the weights are not designed to sum to unity.
In redCV, we use a vector datatype to store the kernel values, for example, identity kernel: [0.0 0.0 0.0 0.0 0.1 0.0 0.0 0.0 0.0]. In redCV, in most filters (calling *rcvConvolve*), the weights are designed to sum to unity and thus the multiply by 1/sumw is not required. However, this factor must be considered if the filter is mixed with other filters to generate a more complex convolution. In this case, the best way is to set the factor parameter to 1/sumw to make the normalization. You can also call *rcvFilter2D* which gives the same result, since the sum of the weights is computed during the summation, and used to scale the result.
The idea here is to make fast image processing with classical filters. In most cases, predefined values are sufficient for a good result. Programs do accept RGB images or matrices, but often transform image to matrix for a faster computation.

Gaussian Filters

Gaussian filters are generally used for smoothing (blurring) images since this kind of filter removes high frequencies present in image. Basically, blurring is done by taking the average of the current pixel and its *n* neighbors.

Source image

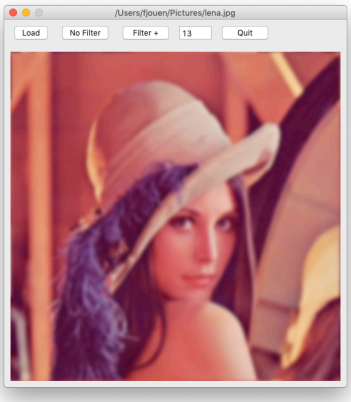
gaussian0.red, takes the sum of the current pixel and its 4 neighbors, and divide it through 5. This allows to create a simple kernel [0.0 0.2 0.0 0.2 0.2 0.2 0.0 0.2 0.0]. Since the sum of weights equals 1.0, you can directly call *rcvConvolve* function. Repeating the convolution process on image makes a larger and larger blurring effect.



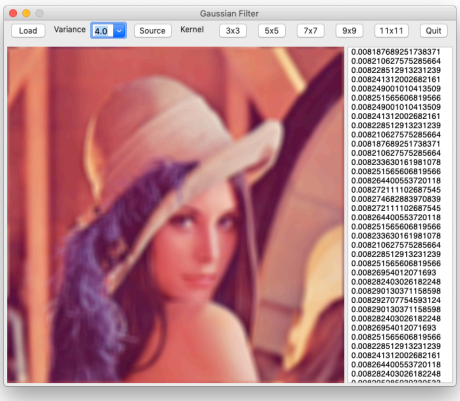
gaussian1.red is similar in results, but directly uses predefined *rcvGaussianFilter* function which first creates a 3x3 uneven Gaussian kernel according to the following equation:

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

where, x is the distance along horizontal axis measured from the origin, y is the distance along vertical axis measured from the origin and σ is the standard deviation of the distribution.
Then *rcvFilter2D* function, which calculates the sum of weights and scales the result, is used.



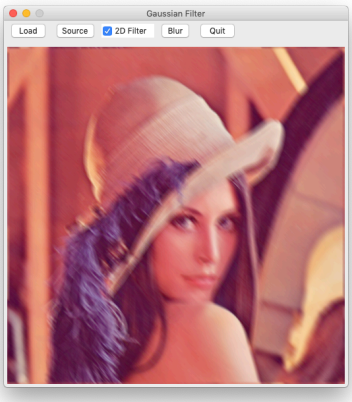
gaussian2.red uses the same algorithm but allows to change the size of the Gaussian kernel, here a 11x11 kernel, and the variance. Calculated kernel values are displayed.



gaussian3.red illustrates how *motion blur* can be easily done with redCV. Motion blur is achieved by blurring in only 1 direction. Here we define a 9x9 motion blur filter

```
knl: [
1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 1.0 0.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0 0.0
0.0 0.0 0.0 0.0 0.0 0.0 0.0 0.0 1.0
]
```

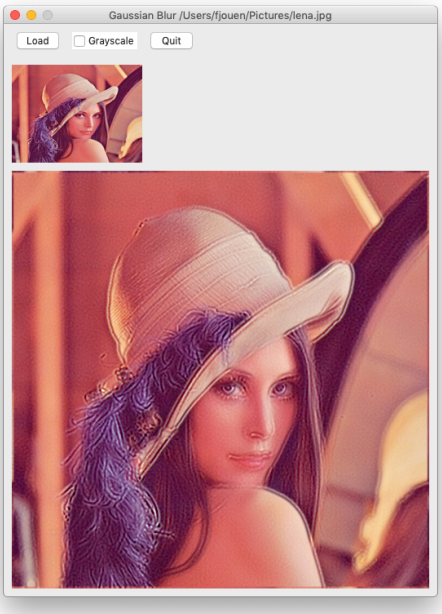
Then the code calls either *rcvFilter2D* or *rcvConvolve* functions. With *rcvConvolve* function we use a factor equals to 1.0/9.0 to normalize the kernel since the sum of weights is different from 1 and equals to 9.



gaussian4.red uses another 7x7 kernel to create uniform motion blur on image**.**

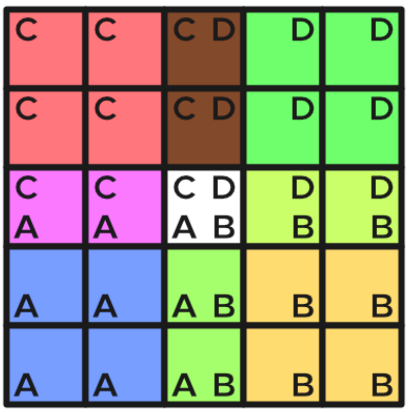
```
knl: [
0.0 0.0 1.0 1.0 1.0 0.0 0.0
0.0 1.0 1.0 1.0 1.0 1.0 0.0
1.0 1.0 -1.0 -1.0 -1.0 1.0 1.0
1.0 1.0 -1.0 -8.0 -1.0 1.0 1.0
1.0 1.0 -1.0 -1.0 -1.0 1.0 1.0
0.0 1.0 1.0 1.0 1.0 1.0 0.0
0.0 0.0 1.0 1.0 1.0 0.0 0.0
]
```

Since the sum of weight equals to 12, we use *rcvFilter2D* function to directly normalize the result and thus, multiplier factor equals to 1.0. You can get the same result with *rcvConvole* function, but in this case, multiplier factor should equal to 1 / 12**.**



Kuwahara Filter

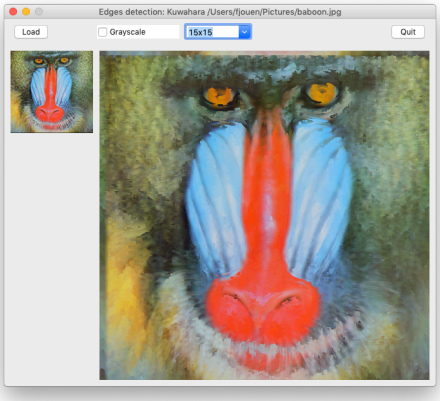
This filter is close to the Gaussian filter. The filter starts with a NxN kernel (where N is odd). Then the kernel is divided into four sub regions [A, B, C, D] as in example, for a 5x5 kernel:



https://github.com/daniel-ilett/smo-shaders

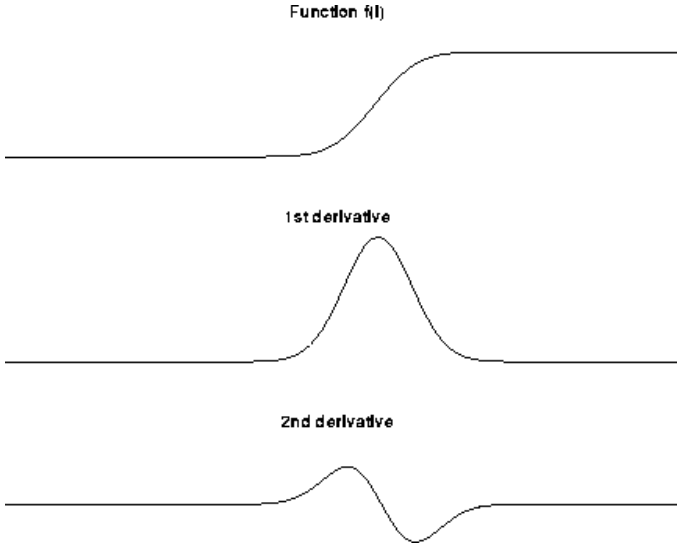
For each pixel corresponding to the center point of the kernel (CDAB), we take the mean value and variance of each sub region. The sub region with the lowest variance will have its mean value used as the replacement value for the current pixel. For RGB images, redCV takes the mean of each RGB color channel and the variance is computed in reference to the image luminance, calculated according to this formula *0.3 R + 0.59 G + 0.11 B***.**

kuwahara1.red and **kuwahara2.red**. This filter can be used for image denoising or adding a nice oil painting effect on the image according to kernel size.



Edges Detection

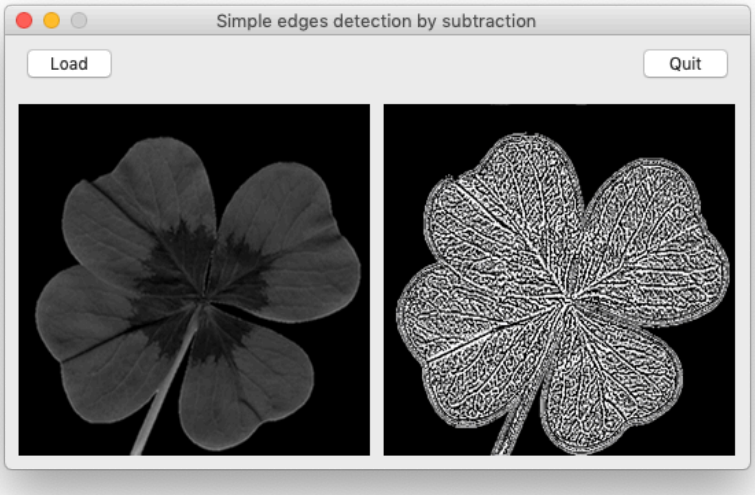
Edges are locations in the image with strong intensity contrast. Since edges often occur at image locations representing object boundaries, edge detection is extensively used in image segmentation to divide the image into areas corresponding to different objects. Since edges correspond to strong illumination gradients, we can highlight them by calculating the derivatives of the image.

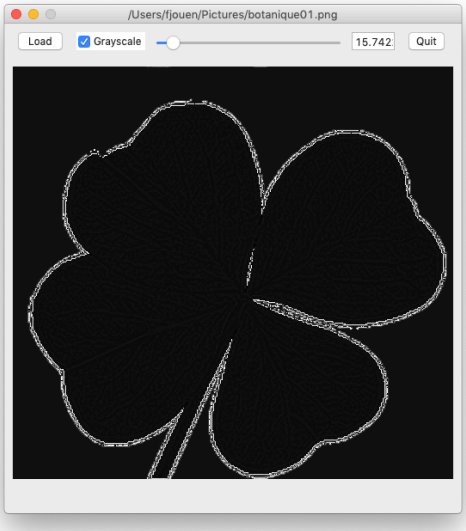


We can see that the position of the edge can be estimated with the maximum of the 1st derivative or with the zero-crossing of the 2nd derivative. You'll find here (https://homepages.inf.ed.ac.uk/rbf/HIPR2/edgdetct.htm) a clear and nice explanation of the technique and the relation between derivative and convolution.

Basic Edges Detection

edges1.red, **edges2.red** and **edges3.red** are very basic filters that use a simple algorithm. First, the original image is smoothed by a 3x3 Gaussian filter and then the smoothed image is subtracted from the original image. **edges3.red** uses a 5x5 kernel filter.





Edges filters

Roberts Filter

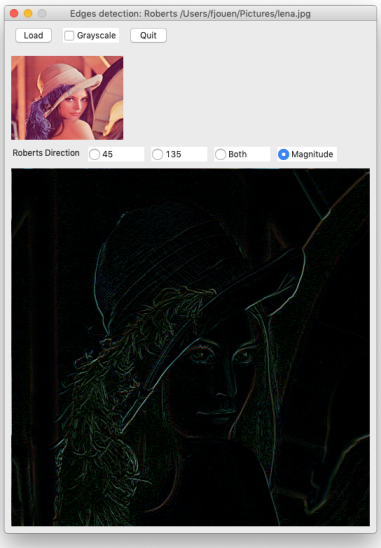
The Roberts filter is one of the oldest and simplest edge detection algorithms to implement. It uses two 2x2 matrices to find the changes in the x and y directions. It computes an approximation of the gradient magnitude of the input image with symmetrical kernels in reference to the diagonal of the kernel matrix. Sum of weights is null.

In **roberts.red**, kernels are defined as follows:

H1: [0.0 1.0 -1.0 0.0] ; 45 °

H2: [1.0 0.0 0.0 -1.0] ;135°

Both h1 and h2 kernels can be combined to calculate the G gradient as explained for Sobel filter. **roberts.red** uses the predefined *rcvRoberts* function.



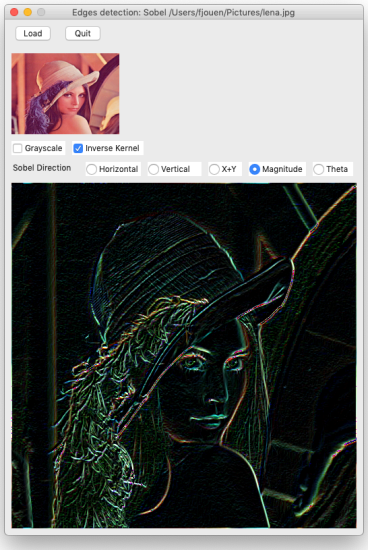
Sobel Filter

The operator calculates the gradient of the image intensity at each point, giving the direction of the largest possible increase from light to dark and the rate of change in that direction. This filter offers symmetrical horizontal and vertical kernels. Signs of weights are opposite and the sum of weights equals to zero.

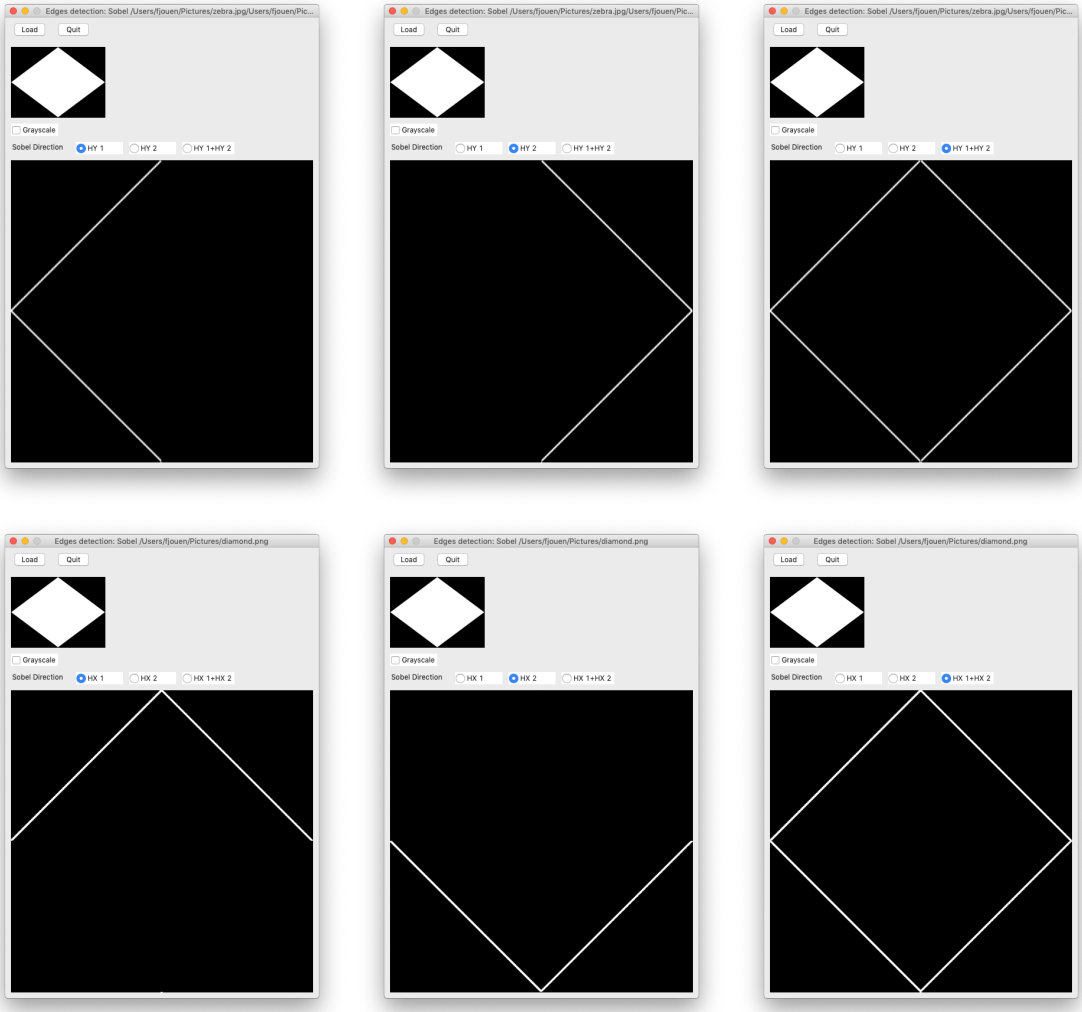
sobel.red includes 4 kernels which correspond to different orientations:

hx1: [-1.0 -2.0 -1.0 0.0 0.0 0.0 1.0 2.0 1.0]
hx2: [1.0 2.0 1.0 0.0 0.0 0.0 -1.0 -2.0 -1.0]
hy1: [-1.0 0.0 1.0 0.0 0.0 0.0 -1.0 0.0 1.0]
hy2: [1.0 0.0 -1.0 0.0 0.0 0.0 1.0 0.0 -1.0]

hx kernels are sensitive to changes in the x direction, i.e., edges that run vertically, or have a vertical component. Similarly, hy kernel are sensitive to changes in y direction. Both hx and hy kernels can be combined to calculate the G gradient. The two gradients computed at each pixel (G_x and G_y) by convolving with above kernels can be considered as the x and y components of gradient vector. This vector is oriented along the direction of change, normal to the direction in which the edge runs. We use this formula for calculating the *magnitude*: *G= sqrt G_x^2 + G_y^2*. To reduce the complexity of above square root calculation, gradient magnitude can also be approximated by absolute summation: *G = abs(G_x) + abs(G_y)*. Lastly the direction Theta is also calculated as atan(G_x/G_y). **sobel.red** uses *rcvSobel* function. Since kernels are symmetrical, you can inverse kernels.



sobel2.red and sobel3.red illustrate how both vertical and both horizontal kernels can be associated to improve edges detection.

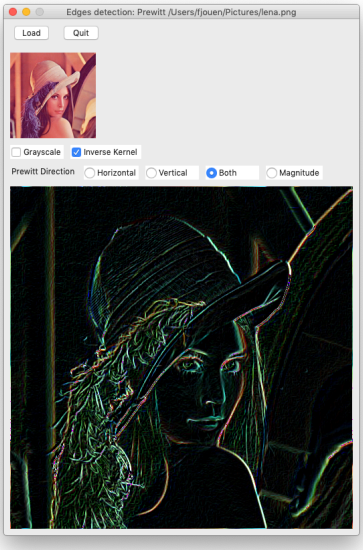


Prewitt filter is an interesting filter with symmetrical horizontal and vertical kernels. Signs of weights are opposite and the sum of weights equals to zero.

prewitt.red uses *rcvPrewitt* function with a parameter which allows to inverse horizontal and vertical kernels.

hx1: [-1.0 0.0 1.0 -1.0 0.0 1.0 -1.0 0.0 1.0]
hx2: [1.0 0.0 -1.0 1.0 0.0 -1.0 1.0 0.0 -1.0]
hy1: [-1.0 -1.0 -1.0 0.0 0.0 0.0 1.0 1.0 1.0]
hy2: [1.0 1.0 1.0 0.0 0.0 0.0 -1.0 -1.0 -1.0]

Both hx and hy kernels can be combined to calculate the G gradient $G= sqrt\ G_x\text{^}2 + G_y\text{^}2$. Since kernels are symmetrical, you can inverse kernels. Both vertical and both horizontal kernels can be associated to improve edges detection.

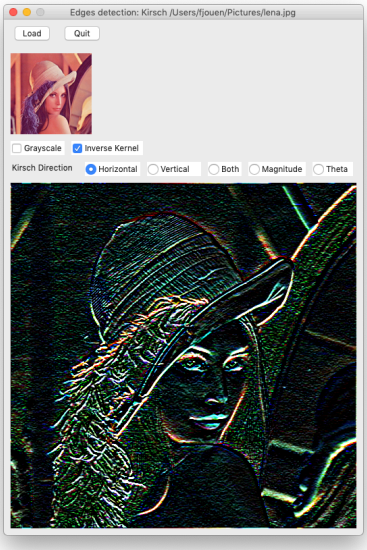


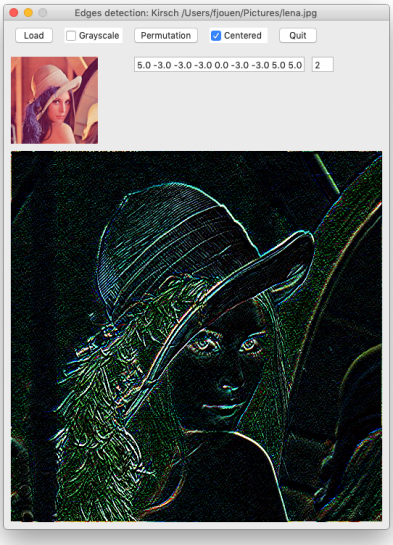
Kirsch filter is also a symmetrical filter with horizontal and vertical kernels.

kirsch.red uses *rcvKirsch*function with a parameter which allows to inverse horizontal and vertical kernels.

hx1: [-3.0 -3.0 5.0 -3.0 0.0 5.0 -3.0 -3.0 5.0]
hy1: [-3.0 -3.0 -3.0 -3.0 0.0 -3.0 5.0 5.0 5.0]
hx2: [5.0 -3.0 -3.0 5.0 0.0 -3.0 5.0 -3.0 -3.0]
hy2: [5.0 5.0 5.0 -3.0 0.0 -3.0 -3.0 -3.0 -3.0]

kirsch2.red uses hY2 kernel to illustrate how to do circular permutation of symmetrical kernels. Permutation can be centered: central kernel value remains unchanged by permutation of other values.

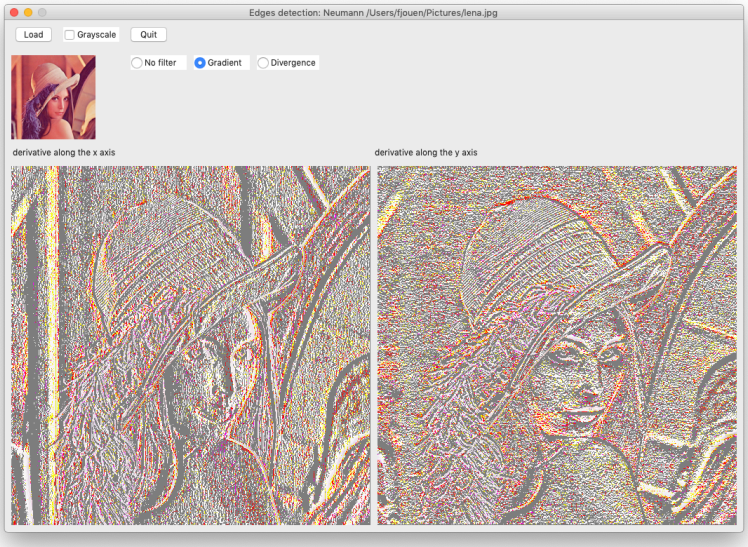




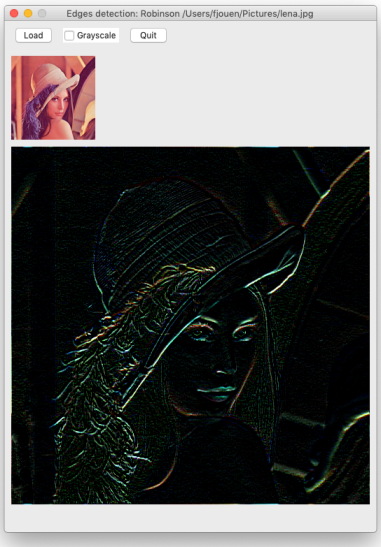
Both vertical and both horizontal kernels can be associated to improve edges detection.

Other Filters

Neuman

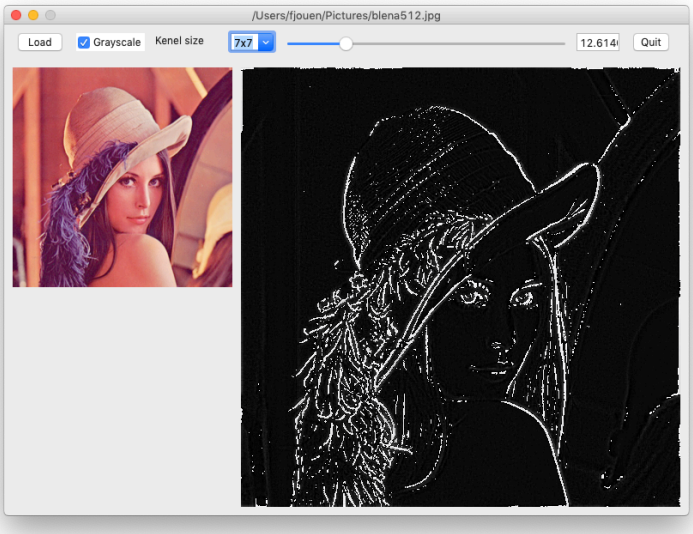


rcvRobinson
knl: [1.0 1.0 1.0 1.0 -2.0 1.0 -1.0 -1.0 -1.0]
robinson2**: allows kernel permutations.**

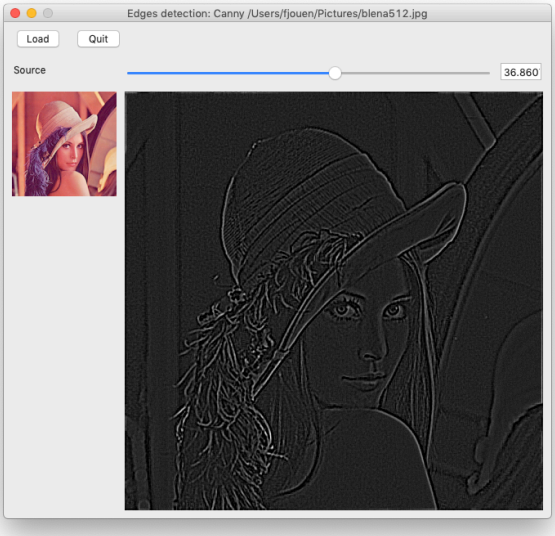


Canny Edge Detection
Canny edge detection is an image processing method used to detect edges in an image while suppressing noise. In redCV, we adopt different ways for implementation.

canny1.red is a basic Canny filter by subtraction (smoothed image - original image). It's works because Gaussian filter + delta function ≈ Laplacian of Gaussian.



canny2.red is more complete and uses 3 successive steps. First, we transform the source image to grayscale image. Then we apply a Gaussian blur with a 3x3 kernel and lastly, we convolve the image with a Laplacian kernel [-1.0 -1.0 -1.0 -1.0 8.0 -1.0 -1.0 -1.0 -1.0].



canny3.red and canny4.red are a complete implementation of Canny algorithm. The Canny edge detector is an edge detection operator that uses a multi-stage algorithm to detect a wide range of edges in images. It was developed by John F. Canny in 1986. You can find here (https://towardsdatascience.com/canny-edge-detection-step-by-step-in-python-computer-vision-b49c3a2d8123) a really clear explanation of the algorithm. Thanks to Sofiane Sahir.
The Canny edge detection algorithm is composed of 5 successive steps:

- Noise reduction
- Gradient computation
- Non-maximum suppression
- Double thresholding
- Edge tracking by hysteresis.

First of all, Canny filter requires grayscale image for a better edges detection which is easily done with *rcvLoadImage/grayscale* function. For the *noise reduction* step, we use a 5x5 Gaussian kernel:

```
knl: rcvMakeGaussian 5x5 1.0
rcvFilter2D cImg dst knl 1.0 0.0
```

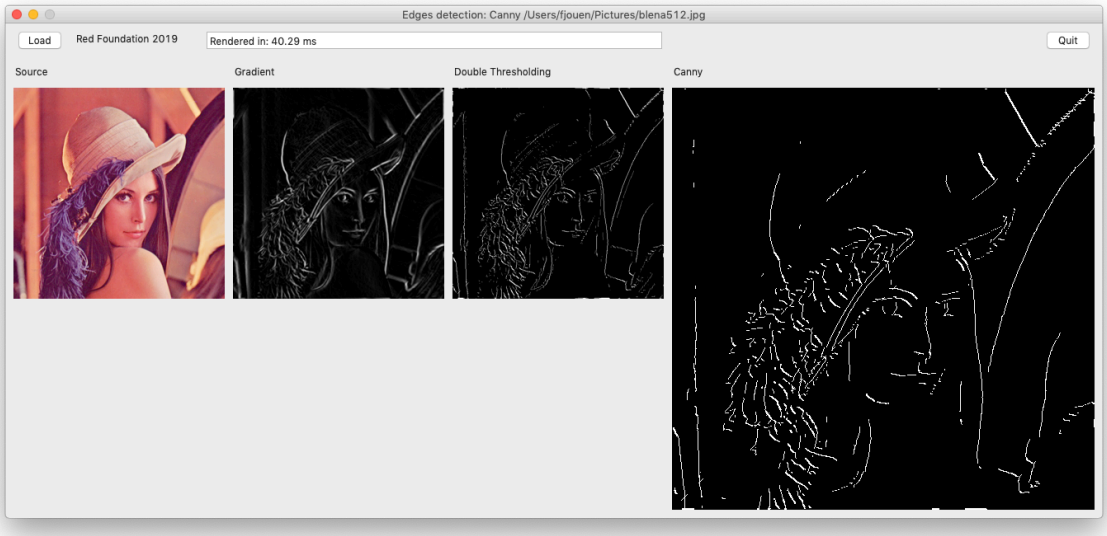
The *Gradient computation step* detects the edge intensity and direction by calculating the gradient of the image using edge detection operators. Edges correspond to a change of pixels intensity. To detect it, the easiest way is to apply filters that highlight this intensity change in both directions: horizontal (derivX) and vertical (derivY). In redCV we use the Sobel filter to get X and Y changes:

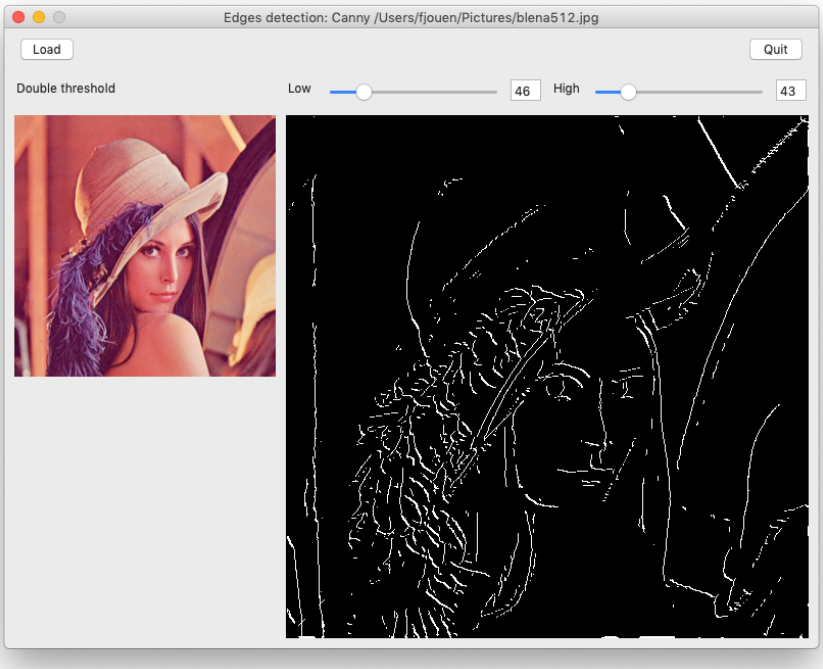
```
hx: [-1.0 0.0 1.0 -2.0 0.0 2.0 -1.0 0.0 1.0]
hy: [1.0 2.0 1.0 0.0 0.0 0.0 -1.0 -2.0 -1.0]
rcvConvolve cImg imgX hx 1.0 0.0
```

Then, the magnitude **G** and the slope **θ** of the gradient are calculated with *rcvEdgesGradient* and *rcvEdgesDirection* functions. Magnitude G is calculated with *Hypot* formula. Hypot is a mathematical function defined to calculate the length of the hypotenuse of a right-angle triangle. It was designed to avoid errors arising due to limited-precision calculations performed on computers. **θ** is equal to *atan2 derivY derivX* and is transformed to degree. Lastly, we apply a normalization of G matrix in order to get intensities between 0 and 255.

Ideally, the final image should have thin edges. Thus, we must perform *non-maximum suppression* step to thin out the edges. The principle is simple: the algorithm goes through all the points on the gradient intensity matrix and finds the pixels with the maximum value in the edge directions. The purpose of the algorithm is to check if the pixels on the same direction (0, 45, 90 and 135 degrees, based on the angle value from the angles matrix) are more or less intense than the ones being processed. This is done with *rcvEdgesSuppress* function. The result is the same image with thinner edges. But some variation regarding the intensity of edges remains in image.

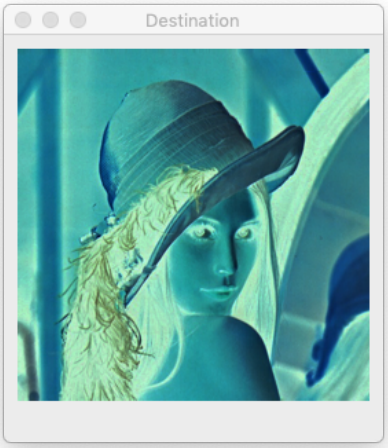
The *double threshold* step aims at identifying 3 kinds of pixels: strong, weak, and non-relevant. High threshold is used to identify the **strong** pixels (intensity higher than the high threshold). Low threshold is used to identify the **non-relevant** pixels (intensity lower than the low threshold). All pixels with intensity between both thresholds are flagged as **weak**.
Based on the thresholding step (*rcvDoubleThresh* function), the *Edge tracking by hysteresis* step (*rcvHysteresis* function) consists of transforming weak pixels into strong pixels, **if and only if one of the pixels around the one being processed is a strong pixel**.





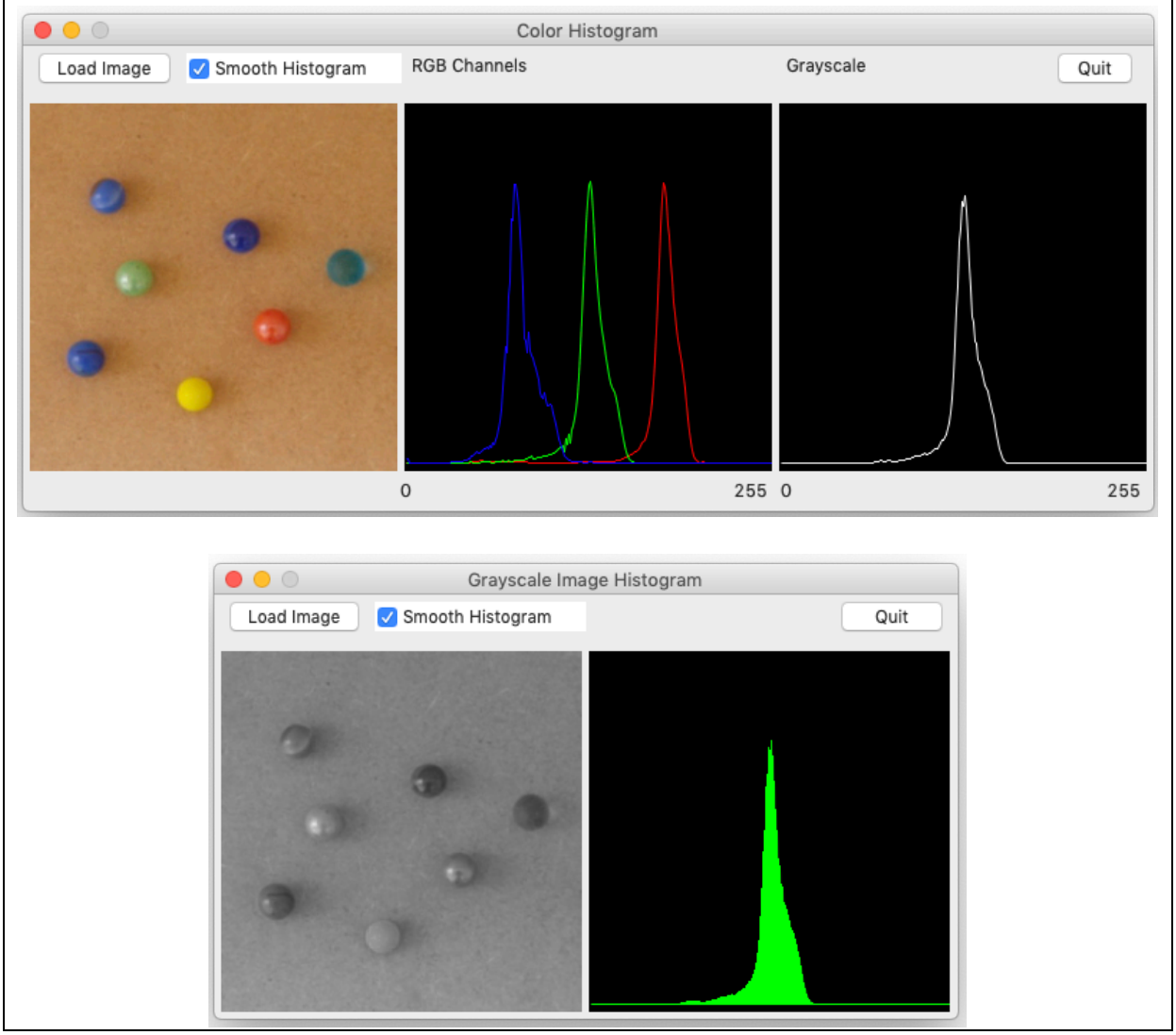
redCV/samples/image_highgui

Code in this directory was developed at the beginning of redCV development in order to test basic functions and probably will disappear in future. However, it remains interesting. **Attention**: you must execute the code from directory and also adapt image location according to your system.
win2.red just shows famous lena.jpg image and **testgui.red** tests some operators.

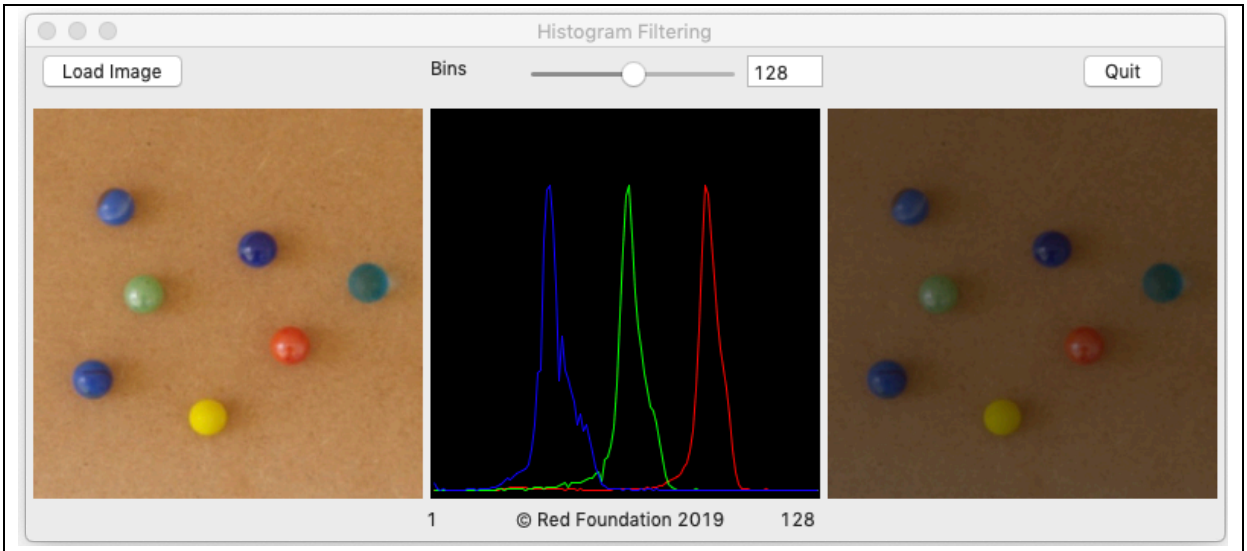


redCV/samples/image_histograms

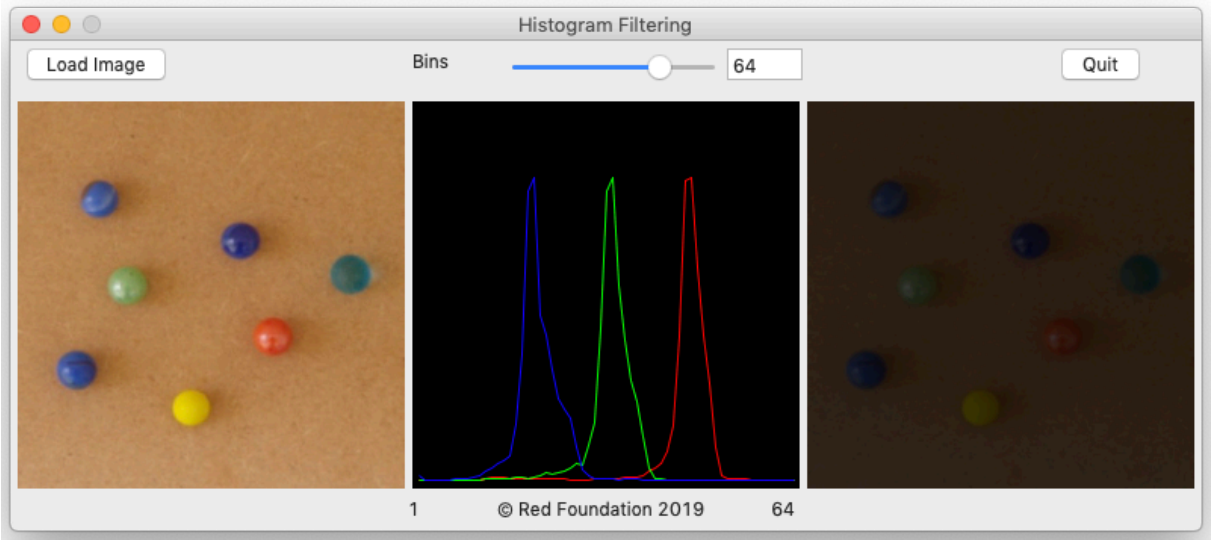
Histograms are useful for identifying RGB components density in image. This is illustrated by **colorHisto.red and grayHisto.red**. In this code, we just calculated the number of RGB component values for each bin [0..255]. This can be done channel by channel or on grayscale image. As illustrated below, RGB values are homogenous across the image.



However, histograms can be used for more sophisticated programs and mainly for clustering color in image. The basic idea is to play with the number of bins and thus to reduce the number of color values. This allows modifications of contrast in image, for example. This is applied in **rgbhiso.red** sample. Code calls *rcvRGBHistogram* routine which makes the job by the application of a divider factor which depends on the number of required bins.



Here, we just use 128 bins rather 256 as usual. The divider equals to 256 / 128 and thus each RGB value will be divided by 2 and affected to the ceiling nearest bin.



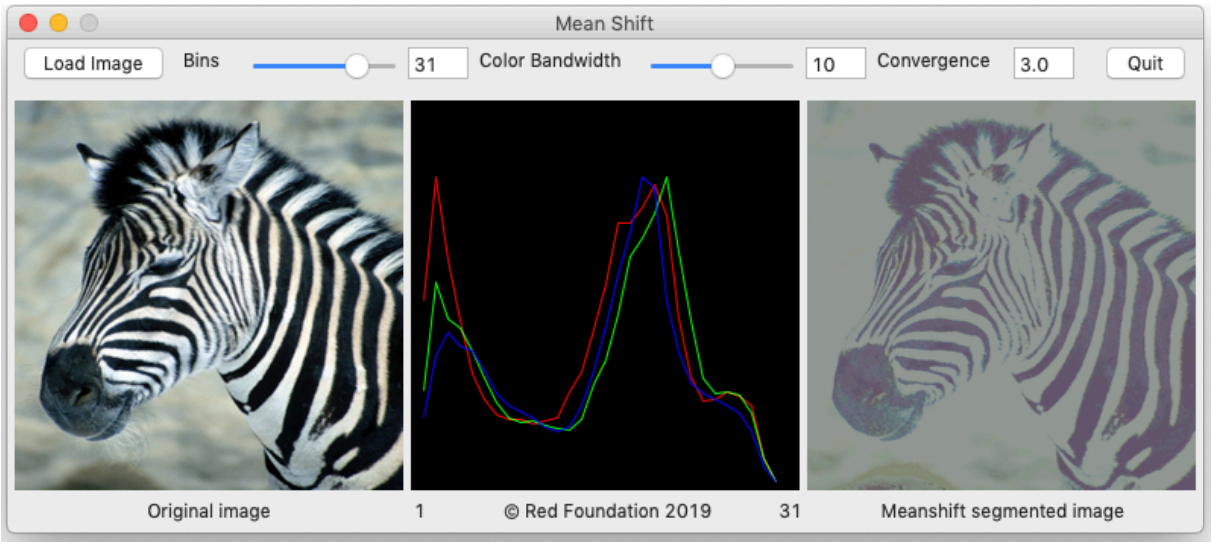
Here, the number of bins is 64 and RGB values are divided by 4 and thus image contrast is reduced.

A similar approach in redCV is the *Mean Shift algorithm*. Mean shift is a standard algorithm used for clustering given set of data points. The steps involved to do image segmentation are simple (**meanShift.red**).

First, you have to compute the histograms of any given RGB image. This is done by the *rcvRGBHistogram* routine included in the lib. Mean shift takes two inputs: the spatial range and the color range. Spatial range (number of bins for RGB image) indicates which pixels in neighborhood are to be considered to compute mean. Color bandwidth indicates pixels of what color in the neighborhood are to be considered.

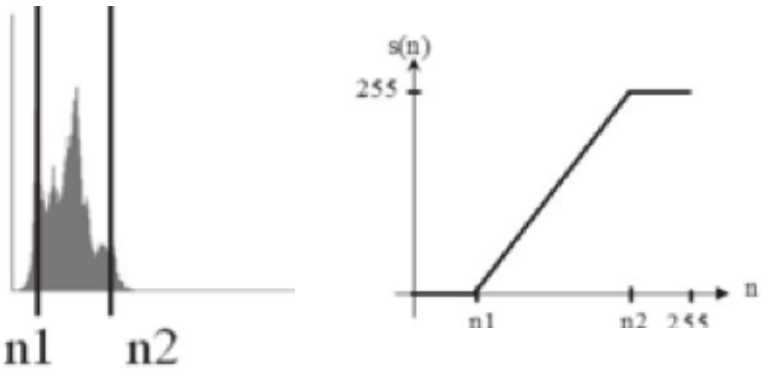
Then, we calculate the mean value for each RGB and we assign the pixel to the mean computed. Since mean shift is iterative we compute the mean again with the new neighborhood and we Iterate this until the mean converges. The convergence is estimated by

measuring the distance between the original bin to which the pixel belonged and the new bin to which it is affected. Lastly, the whole process is repeated for all pixels of image.



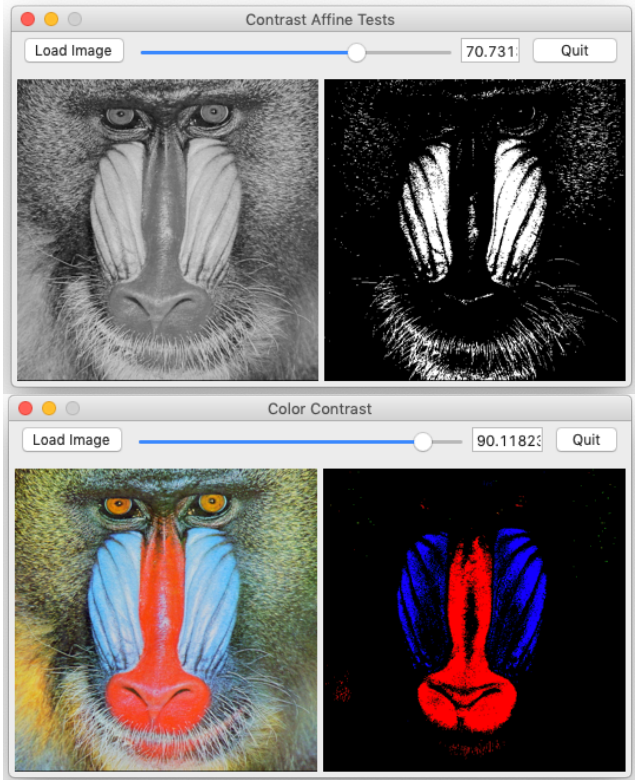
As seen histograms can be used for image segmentation. But histograms can also be efficient for image normalization and image equalization.

Histogram normalization (or dynamic expansion) is a affine transform of gray levels in order that the resulting image use all the dynamical range [0..255]. This is why redCV includes the rcvContrastAffine function which allows to control the gray level and apply an affine transform on pixels as illustrated below.



s(n)= 0 if n<=n1, s(n)= 255 * (n-n1)/(n2-n1) if n1<n<n2 and s(n)=255 if n >= n2

The use of *rcvContrastAffine* function is illustrated in **contrast.red** and **contratsColor.red** code.

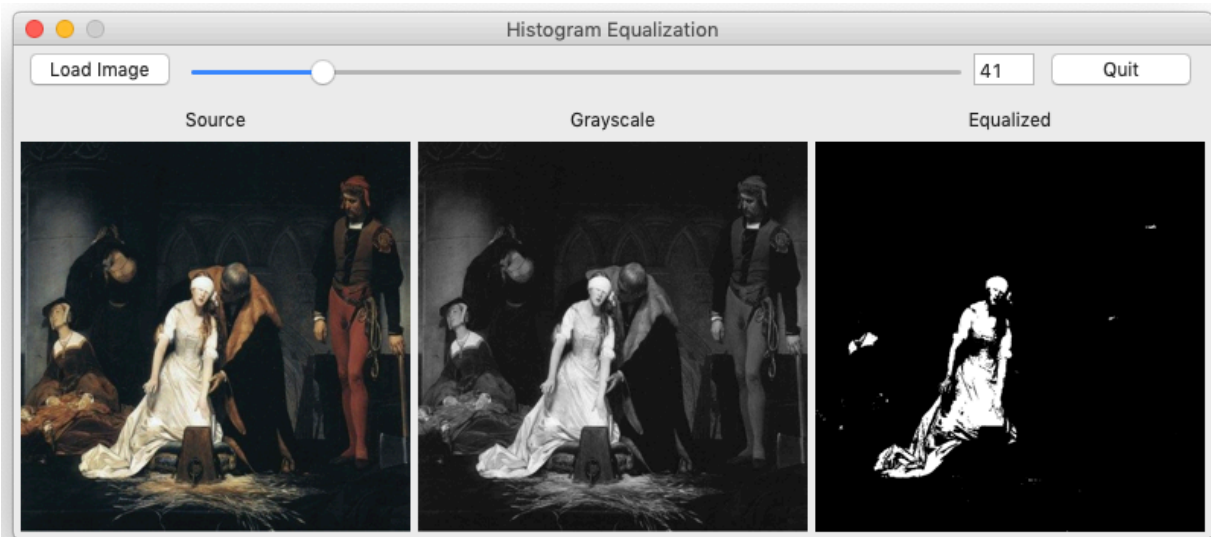


Histogram equalization is a gray level transform with the idea to equilibrate at best the distribution of image pixels across the image dynamic. Ideally, we try to find a flat histogram. *rcvHistogramEqualization* algorithm can be used for this kind of processing (see **equalization.red** or **colorEqualization.red** code samples).

This algorithm performs first the histogram of the source image and then calculates the probability-density function of a pixel value n. *p*(n) is the probability of finding a pixel with the value n in the image. Then the following equation is applied

$$f(a) = D_m \frac{1}{Area_1} \sum_{i=0}^{a} H_c(a)$$

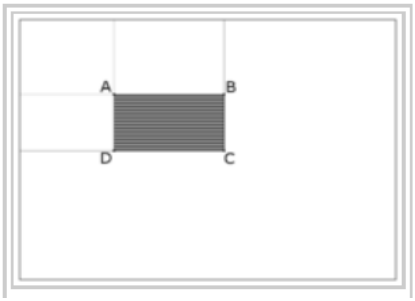
Hc(a) is the histogram of the original image c and Dm is the number of gray levels in the new image b. Area is the size of the image. f(a) simply takes the probability density function for the values in image b and multiplies this by the cumulative density function of the values in image.
This function is useful for improving contrast in low-contrasted images or simply modifying the contrast of image.



redCV/samples/image_integral

The Integral Image (or Summed Area Table) was first introduced in 1984 by Franklin Crow and improved in 2001 by Viola and Jones.

The Integral Image is used as an effective way of calculating the sum pixel values in a given image, or a rectangular subset of the given image. When creating an integral image, we need to create a summed area table. In this table, if we go to any point (x,y) we calculate the sum of all the pixel values **above and to the left** and the original pixel value of (x,y). Using these integral images, one may calculate sum, mean, standard deviation over arbitrary up-right rectangular region of the image in a constant time with only 4 points ABCD. Using integral image, it is also possible to do variable-size image blurring, block correlation...



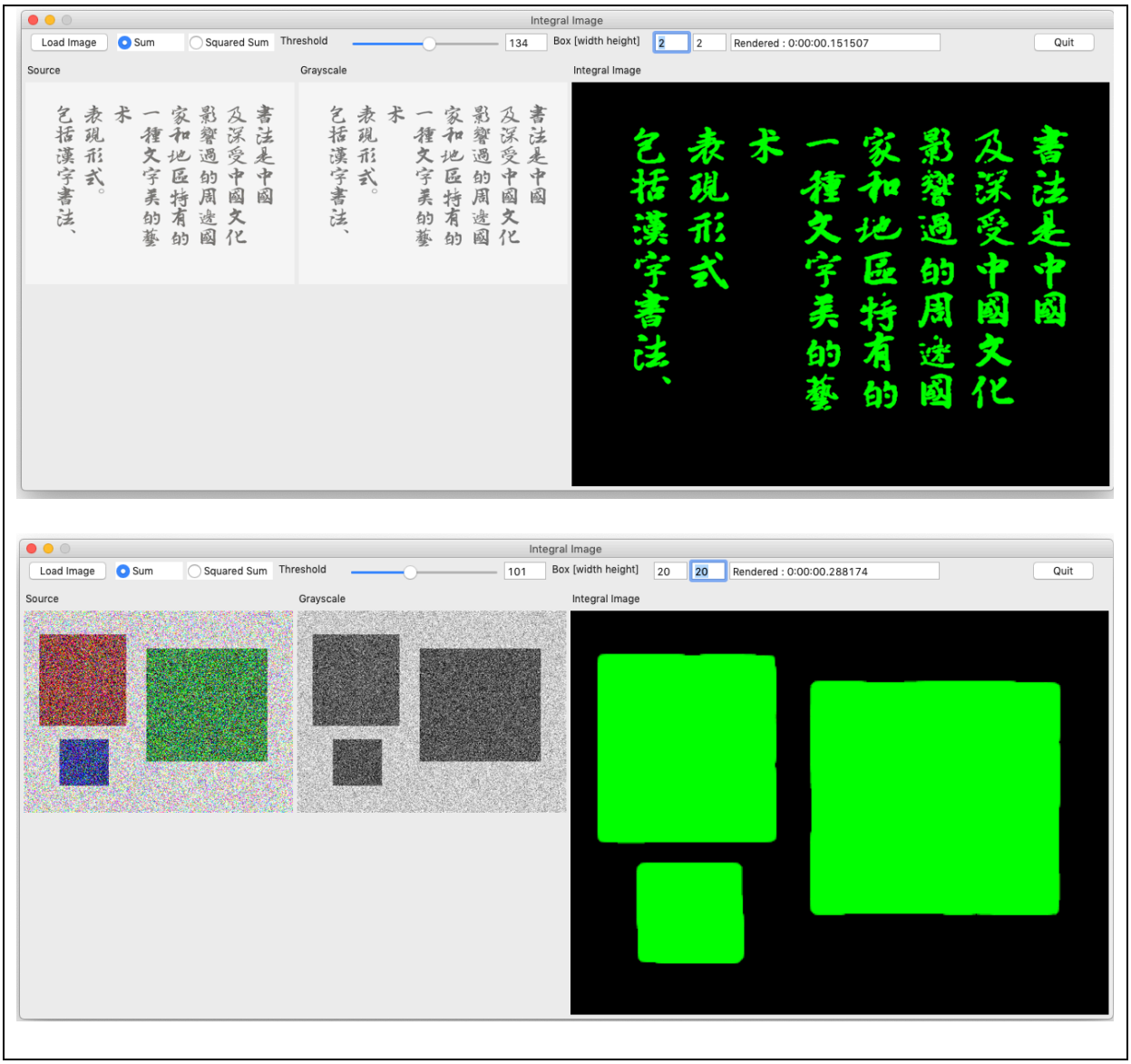
Integral image *ii* is defined according to:

$$ii(x,y) = \sum_{x' \le x, y' \le y} i(x', y')$$

References:
Franklin Crow (1984). «Summed-area tables for texture mapping (http://www.soe.ucsc.edu/classes/cmps160/Fall05/papers/p207-crow.pdf)» SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques: 207–212.

Paul Viola et Michael Jones, Robust Real-time Object Detection (http://research.microsoft.com/~viola/Pubs/Detect/violaJones_IJCV.pdf) IJCV 2001

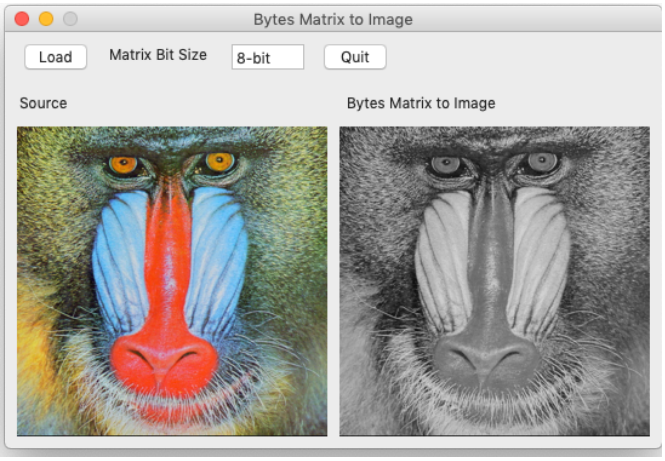
A typical application of integral image is illustrated in **integral2. red** and **integral3.red** for the identification of characters or forms in an image. Integral2.red processes image line by line and is rather slow. Integral3.red calls a routine which includes the direct draw of pairs in the core of the routine. Box height and width are the parameters that define the size of rectangular region to process. Threshold is for filtering the result.


Integral Image
Load Image
Sum
Squared Sum
Threshold
134
Box [width height]
2
2
Rendered : 0:00:00.151507
Quit
Source
Grayscale
Integral Image
書法是中國
及深受中國文化
影響過的周邊國
家和地區特有的
一種文字美的藝
術
表現形式。
包括漢字書法、
Integral Image
Load Image
Sum
Squared Sum
Threshold
101
Box [width height]
20
20
Rendered : 0:00:00.288174
Quit
Source
Grayscale
Integral Image

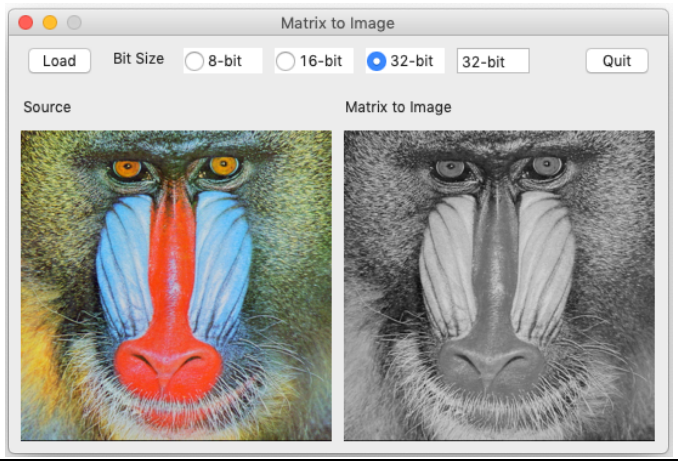
redCV/samples/image_matrices

redCV uses a lot of matrices, which are basically simple Red vectors. A 100 x 100 color image is nothing but an array of 100 x 100 x 3 (for each R, G, B color channel) numbers. Usually, we like to think of 100 x 100 x 3 array as a 3D array, but you can think of it as a long 1D array consisting of 30,000 elements. This is why we use vector! datatype to simulate matrices with Red. Matrices are 2-D with n lines *m columns with only one value. Matrix element can be Char!, Integer! or Float!. RedCV uses integer 8, 16 or 32-bit matrices or 32 or 64-bit float matrices. Matrices are intensively used to simulate 1-channel image for faster rendering.

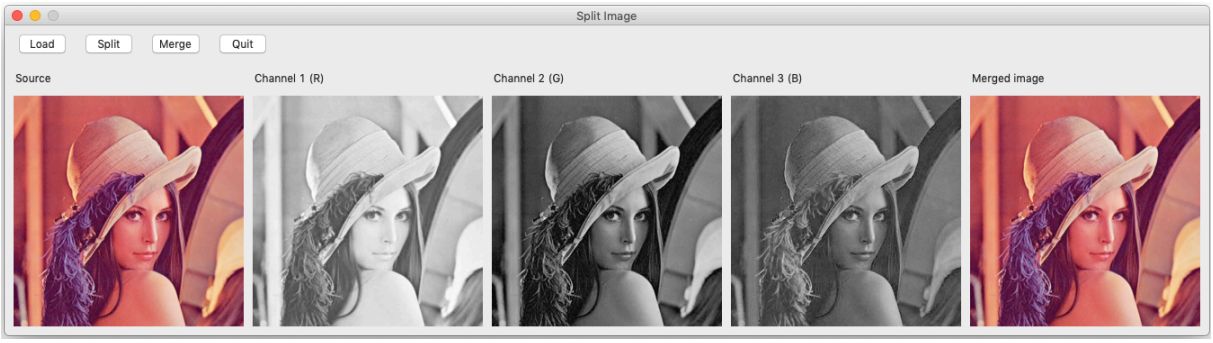
image2Bytes.red is very simple and uses 2 routines: *rcvImage2Mat* and *rcvMat2Image*. rcvImage2Mat creates a grayscale image for any supported Red images and store the result in a 8-bit matrix. rcvMat2Image makes the opposite operation: takes an 8-bit matrix and returns a Red image.



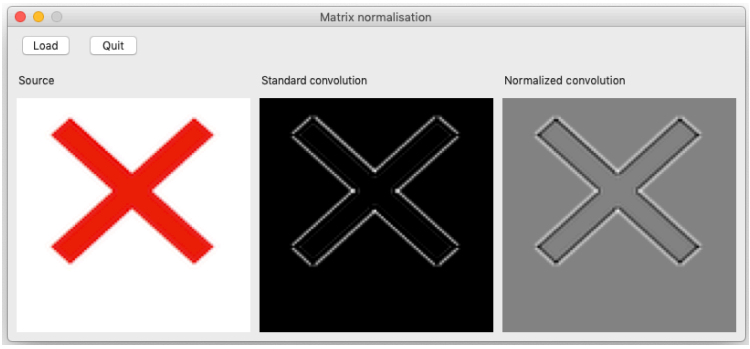
Of course, according to your application, you can use 16 or 32-bit integer matrices. Have a look to **image2Matrix.red** code which use the same functions applied to 8, 16 or 32-bit matrix. As expected, there is no change in image rendering according to the bit size. rcvMat2Image can read any type of integer matrix.



Similarly to images, matrices can be used for splitting and merging as in **splitImage.red** code. This code calls the *rcvSplit2Mat routine* which splits any image to 4 matrices containing A, R, G and B values. Then the code calls the *rcvMerge2Image* routine which creates a new image from the ARGB channels.

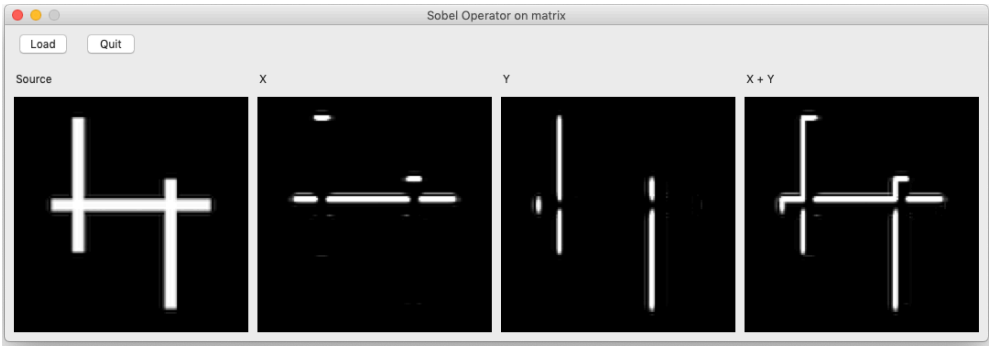


We can also use a normalization on matrix as for image in order to attenuate distortions caused by lights and shadows. In **normalizedMatrices.red**, and in **matLaplacian.red** we compare the effect of a standard Laplacian convolution vs. a normalized Laplacian convolution.

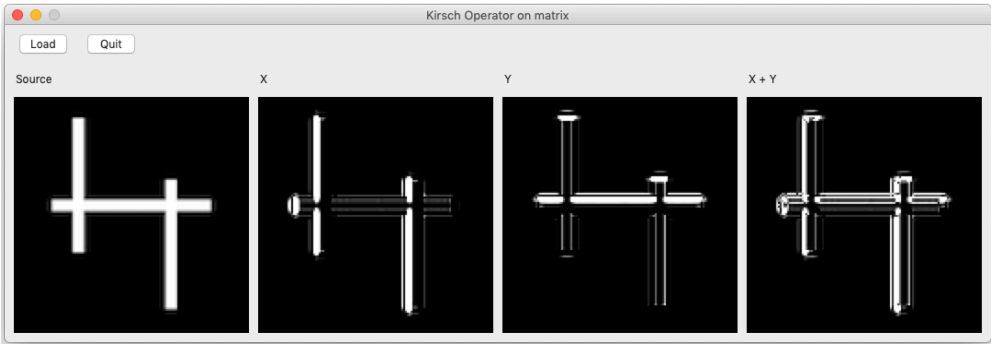


Lastly, you'll find a series of spatial filters directly applied to matrices. For the detail of filters, refer to image_filters section.

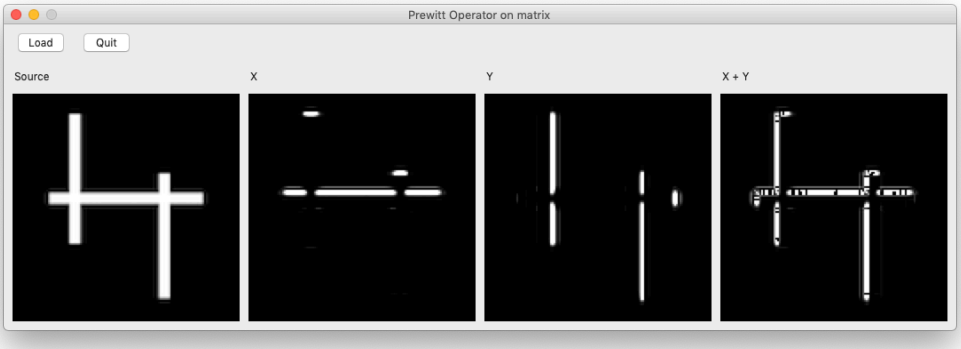
matSobel.red and matFsobel.red apply Sobel operator on integer matrix.



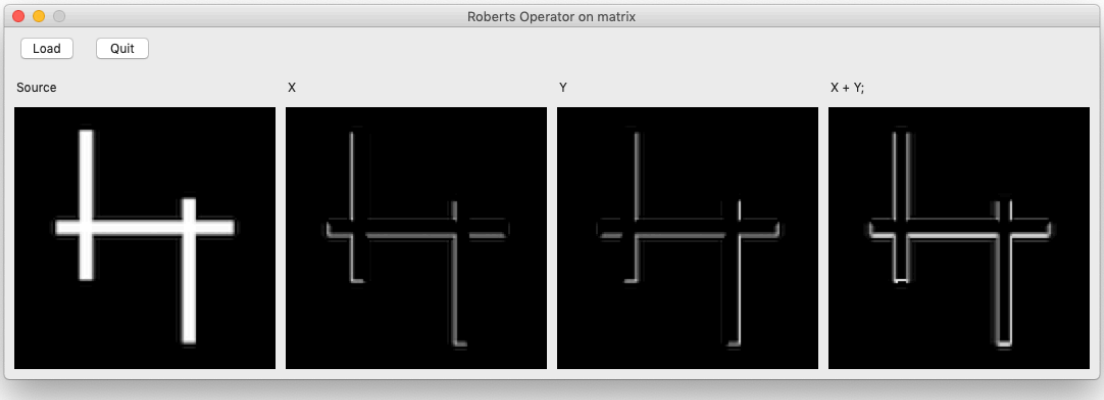
matKirsch.red is for Kirsch operator.



matPrewitt.red for Prewitt operator



matRoberts.red is for the Roberts operator



redCV/samples/image_morphology

Morphological operations were originally developed for bilevel images for shape and structural manipulations. Basic functions are *dilation and erosion*. The association of dilation and erosion in different order results in more high-level operations, such as *closing* and *opening* operators. Morphological operations can be used for smoothing and edge detection or extraction of image features. Morphological operations belong to the category of spatial domain filter. Morphological filters are essentially **set operations**.

This section is based on Yao Wang's teaching (EL5123: Non-linear Filtering) at NYU-Polytechnic, Brooklyn, NY 11201.

Dilatation Operator: erodeDilate.red.

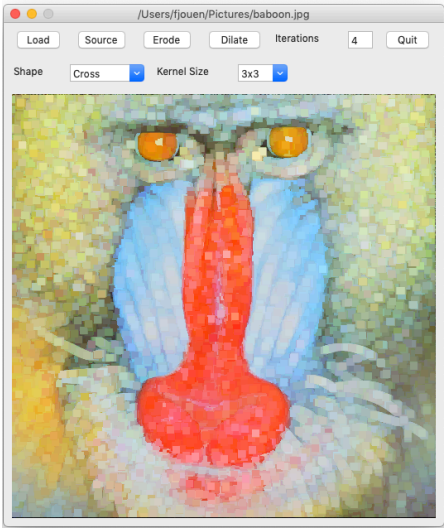
Uses *rcvCreateStructuringElement* and *rcvDilate* functions.

- Dilation of set F with a structuring element H is represented by $F \oplus H$

$$F \oplus H = \{x : (\hat{H})_x \cap F \neq \Phi\}$$

where Φ represent the empty set.

- $G = F \oplus H$ is composed of all the points that when $\hat{H}$ shifts its origin to these points, at least one point of $\hat{H}$ is included in F.
- If the origin of H takes value "1", $F \subset F \oplus H$

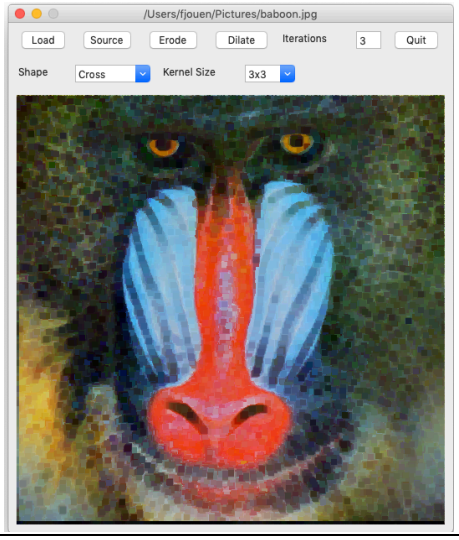


Erosion Operator: erodeDilate.red.
Uses *rcvCreateStructuringElement* and *rcvErode* functions.

- Erosion of set F with a structuring element H is represented by $F\Theta H$, and is defined as,

$$F\Theta H = \{x : (H)_x \subset F\}$$

- $G = F\Theta H$ is composed of points that when H is translated to these points, every point of H is contained in F.
- If the origin of H takes value of "1", $F\Theta H \subset F$



Open/Close operators. **openCose1.red** calls successively *rcvErode* and *rcvDilate* routines or in reverse order for opening and closing operations.
openClose2.red directly uses *rcvOpen* or *rcvClose* functions.

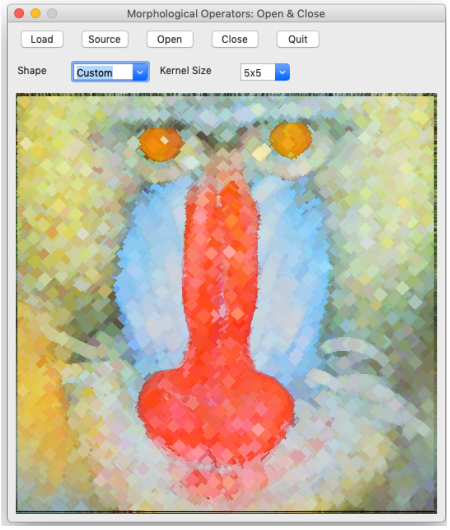
- Closing

$$F \bullet H = (F \oplus H)\Theta H$$

  - Smooth the contour of an image
  - Fill small gaps and holes
- Opening

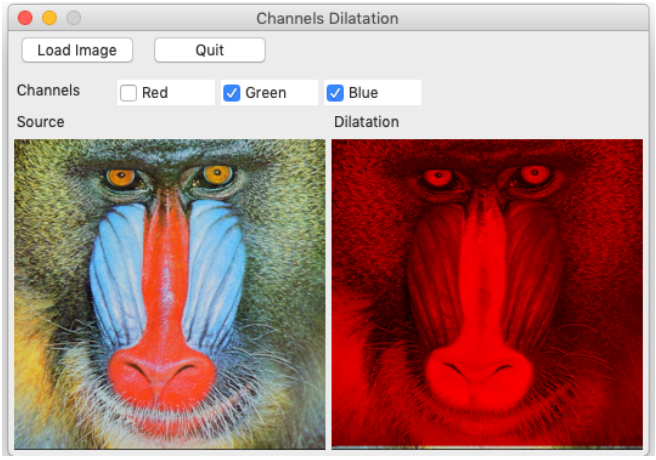
$$F \circ H = (F\Theta H) \oplus H$$

  - Smooth the contour of an image
  - Eliminate false touching, thin ridges and branches



Color erosion and dilatation on RGB channel

Morphological operators presented above directly use RBG values for computation. But with redCV, we can also use these operators for each RGB channel and combine the result. This is illustrated in **colorErosion.red** and **colorDilatation.red** samples. This is useful, if you are looking for specific color values in image.



About structuring element

The shape, size, and orientation of the structuring element depend on application. A symmetrical one will enlarge or shrink the original set in all directions. A vertical one, will only expand or shrink the original set in the vertical direction. A horizontal one, will only expand or shrink the original set in the horizontal direction.

By default, redCV proposes a cross, a rectangle, a vertical or horizontal line as structuring element, but you can create custom structuring elements according to your need such as:

custom: [

0 0 1 0 0

0 1 1 1 0

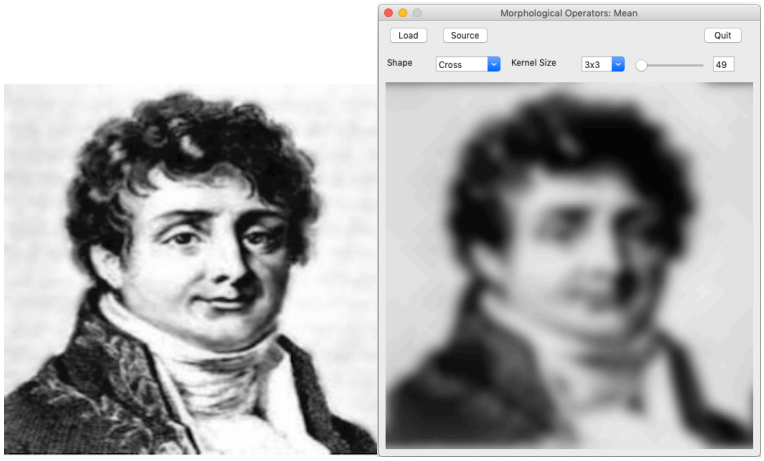
1 1 1 1 1

0 1 1 1 0

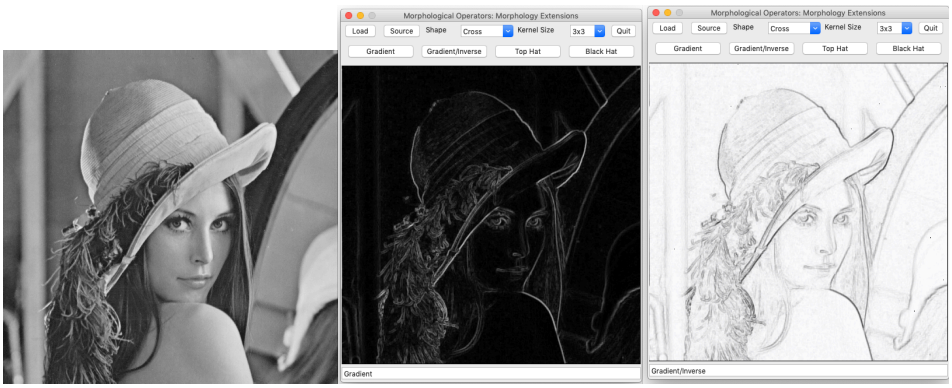
0 0 1 0 0

] which corresponds to a diamond.

Structuring element can be used for a morphological *mean smoothing* as illustrated in **mMean.red** code. In this sample, we can use different kernel shapes and sizes. The basic idea is to calculate the sum of RGB values when kernel value equals to 1, and then calculate the RGB means associated to the current pixel by dividing the RGB sums by the sum of the kernel. This allows a very quick and efficient blurring filtering on image similar to a Gaussian filter.



Morphological operators can also be combined to create really sophisticated image filters. An example is given in **morphologyExt.red** code. This code is written for **grayscale images**. Different filters (*rcvMGradient*, *rcvTopHat* and *rcvBlackHat*), that are using morphological operators, are applied on source image. Then, the resulting image is subtracted from the source image.

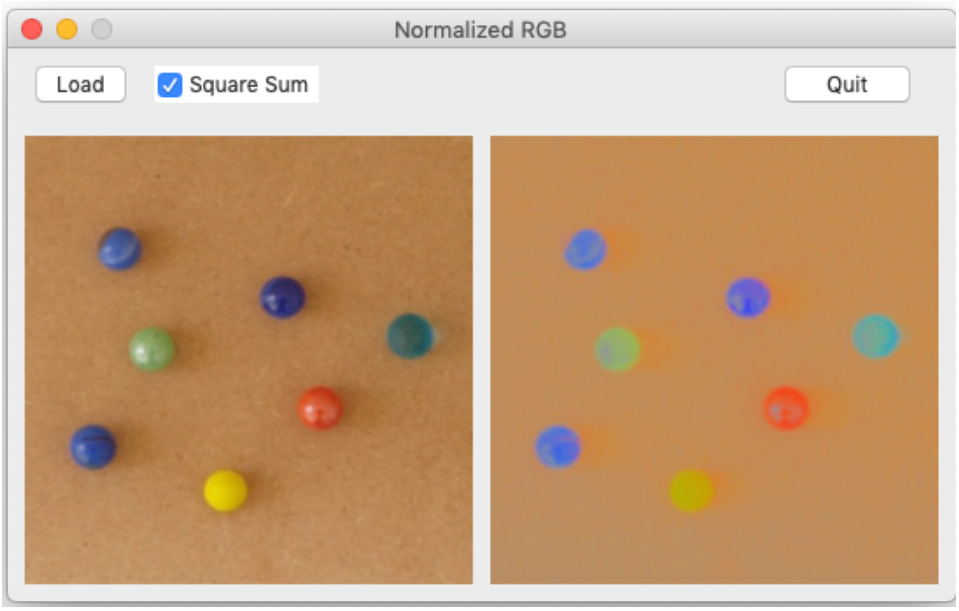


source image rcvGradient rcvGradient/inverse

redCV/samples/image_operators

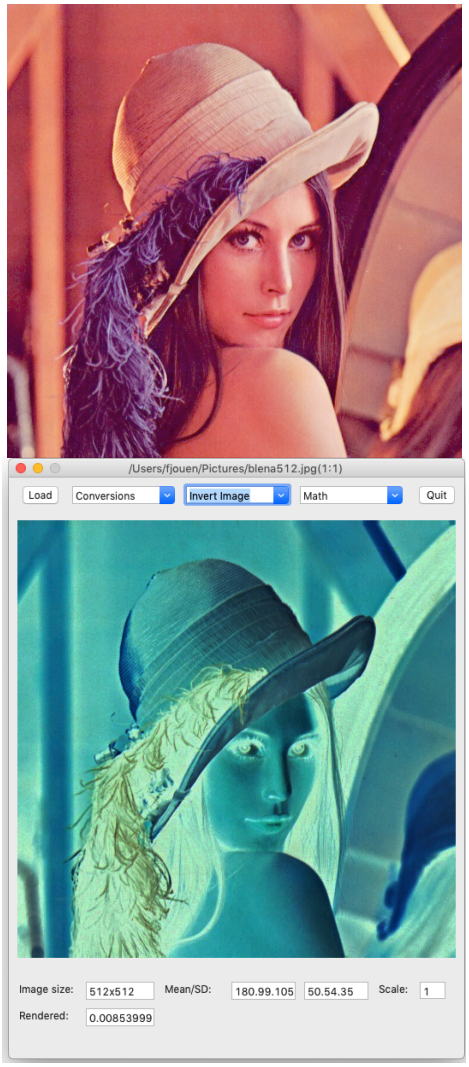
These samples are trivial, illustrate very basic image processing with redCV and, were written at the beginning of redCV development. But some functions are really useful.

normalizedRGB.red.
Sometimes, you want to get rid of distortions caused by lights and shadows in an image. Normalizing the RGB values of an image can be a simple and effective way of achieving this objective. When normalizing the RGB values of an image, you divide each pixel value by the sum of the pixel value over all channels. So, if you have a pixel with intensities R, G, and B in the respective channels, its normalized values will be R/S, G/S and B/S (where, S=R+G+B).



As illustrated here, image background is now uniform and dots color are easier to be detected, since shadows and lights are attenuated.

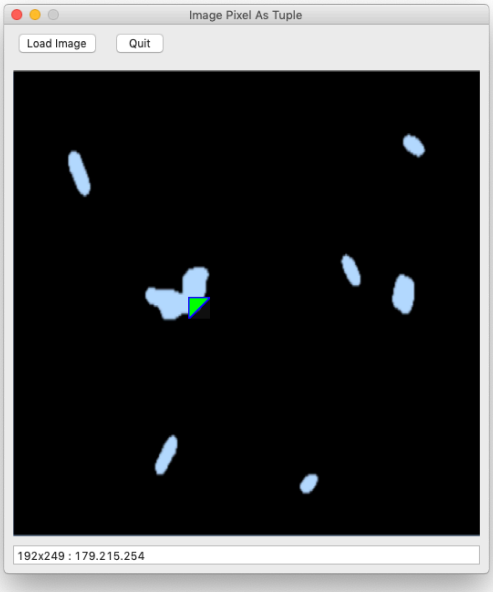
matOps.red, logicalOps.red and opimage.red give some examples of mathematical and logical operators on image with redCV. These samples also included some color conversions.

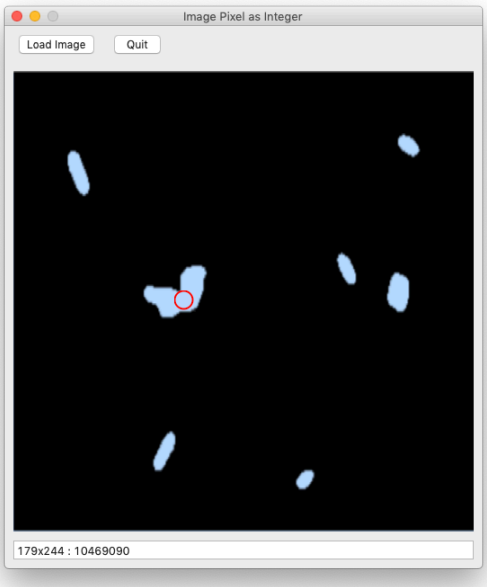


redCV/samples/image_pixels

Code in this directory calls basic but useful functions that allows to read or to write pixels values according to the XY coordinates.

pixel1.red and **pixel2.red** use *rcvGetPixel* and *rcvGetPixelAsInteger* that give access to pixel value either as a tuple or as an integer. *rcvPickPixel* is similar to *rcvGetPixel.*





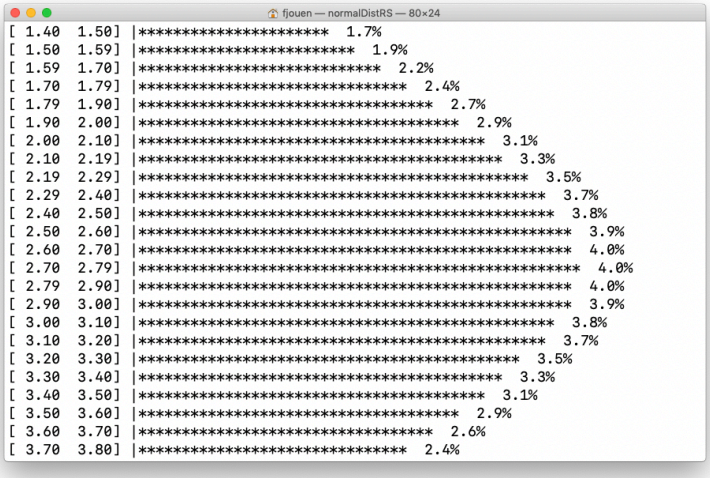
wpixel.red does the opposite operation for writing pixel value by using *rcvSetPixel* or *rcvPokePixel* functions.

redCV/samples/image_random

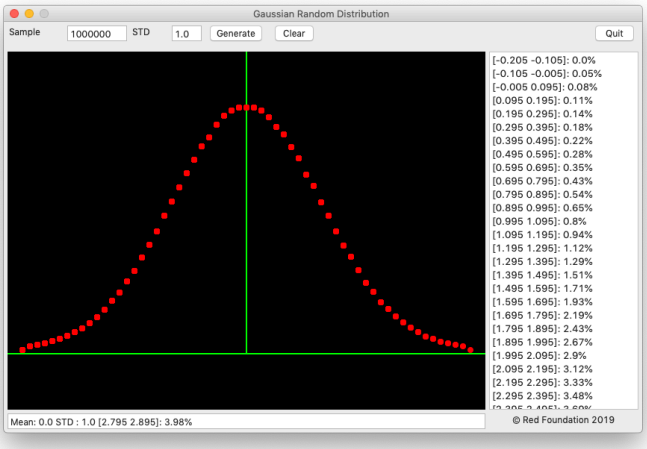
In this directory, you'll find a lot of experiment about random generators which are often need when creating random values for clustering algorithms.

A first series concerns the Red version of generating normal Gaussian distribution with a mean = 0.0 and a standard deviation = 1.0.

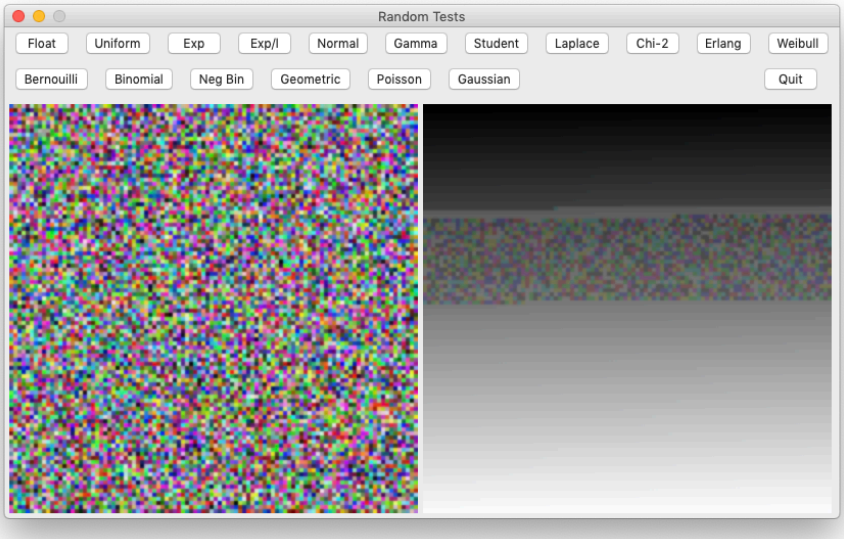
normalDis.red and **nomalDisRS.red** are based on Rosetta Code (Rosetta Code) and do not require view. One code is 100% Red code and the other uses Red/System code for faster rendering, since we are generating 10.000.000 random values.



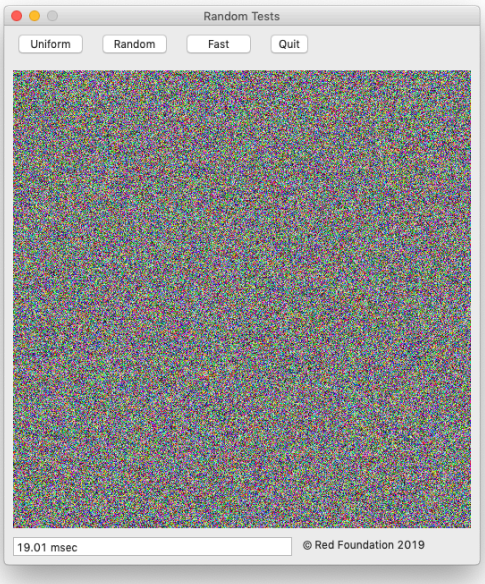
randomView.red is a graphical version that also uses Red/System code for faster computation.



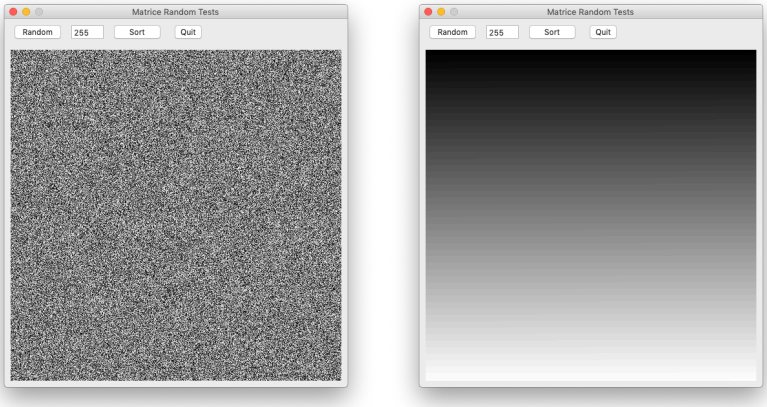
RandomLaws.red allows to test continuous or discrete random generators according to your own use.



random.red uses Red random function to create random images. random.red calls *rcvRandomImage* function with 2 parameters: the image size (a pair!) and the random value as a tuple!. *rcvRandomImage* also uses a refinement.
rcvRandomImage/uniform: generates a random color similar for all image pixels.
rcvRandomImage/alea: generates a different random color for all image pixels.
rcvRandomImage/fast: similar to /alea by calls a Red/System routine for faster rendering.



randomMat.red uses *rcvRandomMat* which is similar to rcvRandomImage/fast but applied to a 1-Channel matrix. **randomMat** also uses *rcvSortMat* for sorting values.

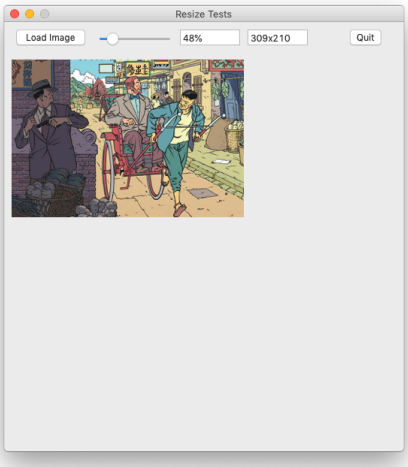


redCV/samples/image_resizing

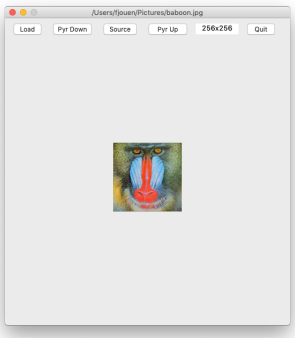
Red OS-IMAGE is now functional on macOS. Thanks to Red Team.

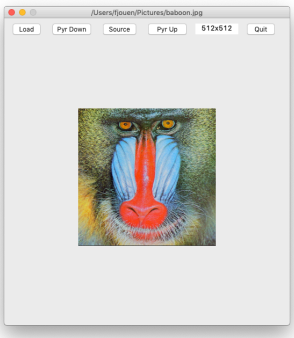
```
_rcvResize: routine [src [image!] w [integer!] h [integer!] return: [image!]
][
        image/resize src w h
]
```

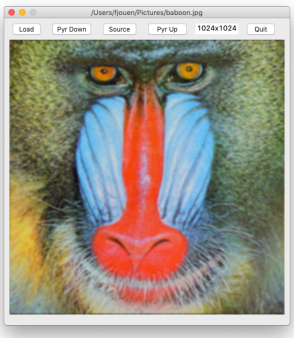
/image_resizing/resize1.red and resize2.red illustrate the use of *rcvResizeImage* function.



redCV also includes (*/image_resizing/pyramidal.red*) 2 important resizing functions for image processing: *rcvPyrDown* and *rcvPyrUp*. *rcvPyrDown* divides the image source size by 2 and the applies a 5x5 Gaussian filter on the image. *rcvPyrUpn* multiplies the image source size by 2 and the applies a 5x5 Gaussian filter on the image. This creates a blurring effect on image, which is very important when we are looking for forms or edges in image. Of course, these functions can be recursively called in order to increase the blurring effect. From general point of view, downsizing source image size will improve the speed of image processing.







Lastly, redCV includes also some draw functions we can use for image resizing. The idea is to use draw commands.

```
lt: 0x0                 ; left-top pixel
br: 0x0                 ; bottom-right pixel
isize:                  ; image size
br: isize
drawBlk: compose [image (dst) (lt) (br)]    ;make a list of draw commands
canvas/draw: drawBlk                        ;show image
isize: isize / 2                            ;modify image size
dst: rcvResizeImage/gaussian dst iSize      ; call cvResizeImage function
drawBlk/4: iSize                            ;update draw block
```

Tip: By the way, macOS users can call **sips** command to resize images: *sips -Z 500 *.jpg //* resample to a max of 500 px for the Y axis. This is easy to include in Red code: *call/wait rejoin ["sips -Z 500 '" to-string imageName "'"]*. Red *to-image* function allows to save the modified image.

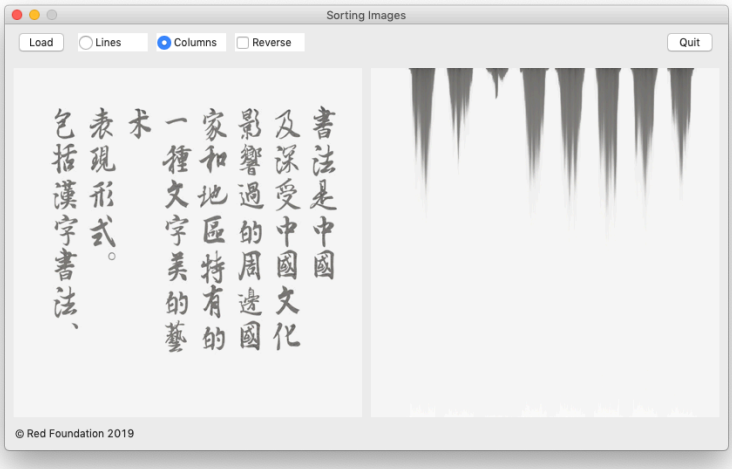
redCV/samples/image_sort

Image sorting is sometimes important for identification of shapes or regions of interest in images. Of course, sorting rgb values from 0.0.0 to 255.255.255 is really easy to do with *rcvSortImage* but not very informative in many cases.

For this raison, *rcvXSortImage* and *rcvYSortImage* functions allow to sort image by lines or columns. Both functions also used a Boolean flag for reversing the image sorting.

sort.red illustrates the use of both functions. Very often, I have to find and to count lines or columns of text in images.

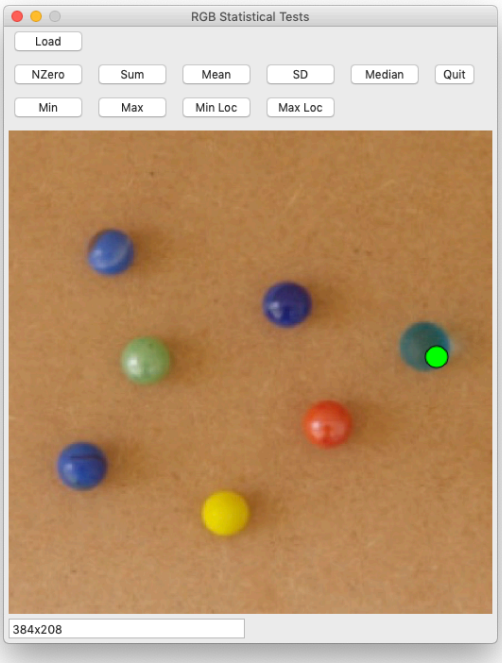
rcvXSortImage finds 7 lines



rcvYSortImage finds 8 columns

redCV/samples/image_statistics

redCV is able to calculate elementary statistics on images and matrices.
imageSats.red uses such predefined functions
rcvCountNonZero: Number of pixels <> 0.0.0
rcvSum: Pixels Sum
rcvMean:Pixels mean
rcvSTD:Pixels standard deviation
rcvMedian: Pixels median
rcvMinValue: Minimum pixel value
rcvMaxValue : Maximum pixel value
rcvMinLoc: XY position of minimum
rcvMaxLoc :XY position of maximum
matStats.red performs the same statistics on matrix (1 channel image).
Several functions accept */argb refinement* to get computation for ARGB values.

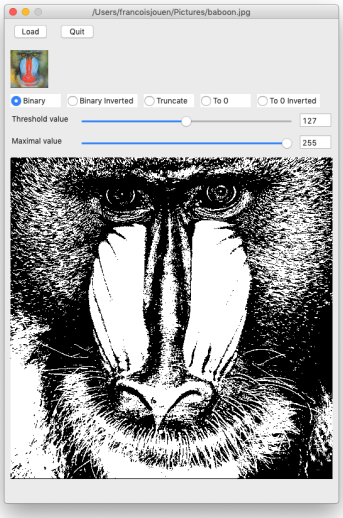


Green dot indicates XY location of minimum RGB value in image.

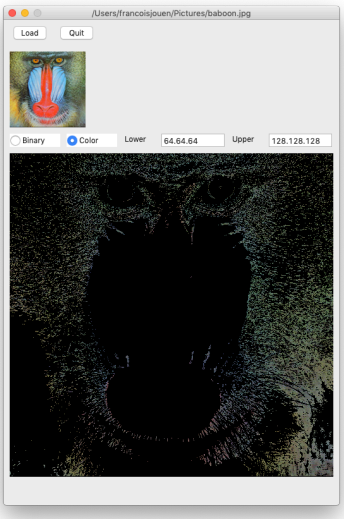
redCV/samples/image_thresholding

redCV includes a series of routines and function for binary or grayscale image thresholding. **Thresholding1.red** and **thresholding2.red** use 2 parameters (minimal and maximal threshold values) and propose different refinements:

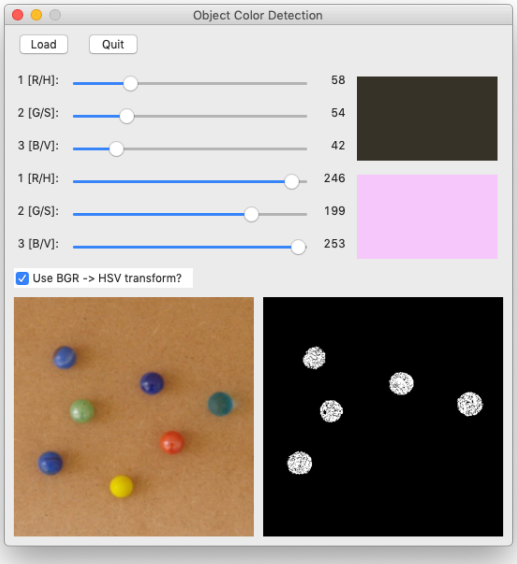
binary: dst(x,y) = maximal value , if src(x,y) > minimal value, 0, otherwise
binaryInv (binary inversion): dst(x,y) = 0, if src(x,y) > minimal value, maximal value, otherwise
trunc: dst(x,y) = minimal value , if src(x,y) >minimal value , src(x,y), otherwise
toZero: dst(x,y) = src(x,y), if src(x,y) >minimal value, 0, otherwise
toZeroInv (zero inversion): dst(x,y) = 0, if src(x,y) > minimal value, src(x,y), otherwise



inRange.red can be applied to color images and allows to select the range of RGB values we want to keep in the destination image.



Lastly, **colorObject.red** demonstrates how more sophisticated color thresholding can be done with redCV. Here the idea is to use HSV color transformation in order to improve shape detection by color. In this sample, low and high threshold values are fixed to identify green and blue dots and to exclude yellow and red objects.



redCV/samples/image_tiff

May be Red will support tiff images in next future, but actually we have to find specific solutions. redCV Tag Image File Format decoder is based on Rev 6.0 from: Developer's Desk Aldus Corporation. The TIFF file format was designed jointly by Aldus and Microsoft with leading scanner vendors to facilitate incorporating scanned images into publishing. This is an interesting format since many information about the image can be included in the file. However, reading or writing tiff files is a little bit more complicated than standard image format since image data can be everywhere in the file.

Basic objects for reading Tiff Files

A TIFF file begins with an 8-byte *image file header* that points to an *image file directory* (IFD). An IFD contains information about the image, as well as pointers to the actual image data.

First, we defined objects for **image file header**:

```
IFFHeader: make object! [
        tiffBOrder:             integer!   ; 2 bytes 0-1 byte order
        tiffVersion:            integer!   ; 2 bytes 2-3 42 version number
        tiffFIFD:       integer!   ; 4 bytes 4-7 offset of the first Image File Directory
]
```

and for entries **associated to image file directory**:

```
TImgFDEntry: make object![
        tiffTag:        integer! ; byte 0-1 TIFF Field Tag 2 bytes 0-1 see TIFF Tag Definitions
        tiffDataType:   integer! ; byte 2-3  Field data type  2 bytes 2-3 see TIFFDataType
        tiffDataLength: integer! ; byte 4-7  number of values of the indicated type; length in spec  4 bytes 4-7
        tiffOffset:     integer! ; byte 8-11 byte offset to field data or value of the field if datalength < 4 bytes
        redValue:       string!   ; supplementary red field to get the "real" value
]
```

redvalue field was added in order to directly store in the object the pointed value of the tag.

We also implemented different objects for processing the various image type supported by TIFF:

- Baseline TIFF bilevel images
- Baseline TIFF grayscale images
- Baseline TIFF palette-color images
- Baseline TIFF RGB image

These objects are defined in redCV/libs/tiff/redCVTiff.red.

Reading Image File Header

As underlined a TIFF file begins with an 8-byte image file header, containing the following information. **Bytes 0-1**: The byte order used within the file. This is the identification, 'II' stands for Intel byte order, 'MM' for Motorola byte order. The following data must be interpreted accordingly!

This why is included in redCV a short function to reverse the byte order of the binary values for Motorola files.

```
bigEndian: false ; by default intel
endian: func [str [binary!]] [either not bigEndian [to-integer reverse str] [to-integer str]]
```

Bytes 2-3: An arbitrary but carefully chosen number (42) that further identifies the file as a TIFF file. **Bytes 4-7**: The offset (in bytes) of the first image directory. The first image directory must begin on an even byte boundary. The image directory may follow or not the image data it describes.
Reading tiff file header is really simple with redCV. You must first open the file as binary! and just read 8 bytes at the onset of the file with this kind of function.

```
rcvReadTiffHeader:  func [] [
        tiff: head tiff
        tmp: to-string copy/part skip tiff 0 2 ; byte order
        ;file created by motorola (big endian) or intel (litte endian) processor?
        either (tmp = "MM") [bigEndian: true byteOrder: "Motorola"]
                            [bigEndian: false byteOrder: "Intel"]
        TiffHeader/tiffBOrder: tmp
        tmp: copy/part skip tiff 2 2
        ;if  endian  tmp <> 42 [exit] ; mettre test ici
        TiffHeader/tiffVersion: endian tmp  ; expected value: 42
        tmp: copy/part skip tiff 4 4
        TiffHeader/tiffFIFD: endian tmp ;offset of the first Image File Directory
]
```

Reading Image File Directory
An Image File Directory (IFD) consists of two bytes indicating the number of entries followed by the entries themselves. The IFD is terminated with 4 bytes offset to the next IFD or 0 if there are none. A TIFF file must contain at least one IFD. In case of multiple images there is an IFD for each image included in the file. So the second operation to do with Red is to make the list of IFDs present in the file as explain in the following function.

```
rcvmakeTiffIFDList: func [] [
   IFDOffsetList: copy []
   ;move to the First Image directory offset and get the number of entries
   tiff: head tiff
   stream: skip tiff TiffHeader/tiffFIFD
   startOffset: TiffHeader/tiffFIFD
   tmp: copy/part stream 2
   numberOfEntries:  endian  tmp     ; number of directory entries
   bloc: copy []
   append bloc startOffset
   append bloc  numberOfEntries
   append/only IFDOffsetList  bloc
   ;move to the next IDF offset and get the value
   nextOffset: startOffset + 2 + (numberOfEntries * 12)      ; Offset of next IFD
   stream: skip tiff nextOffset
   tmp: copy/part stream 4
   offsetValue:  endian  tmp
   ;now move to the other IFDs if exist and get the number of entries
   ; this is case for multi pages files
   ;repeat until ifd offset = 0
   if  offsetValue > 0 [
           until [
                   startOffset: offsetValue
                   stream: skip tiff startOffset
                   tmp: copy/part stream 2
                   numberOfEntries: endian tmp
                   bloc: copy []
                   append bloc startOffset
                   append bloc numberOfEntries
                   append/only IFDOffsetList bloc
```

```
                    nextOffset: startOffset + 2 + (numberOfEntries * 12)
                    stream: skip tiff nextOffset
                    tmp: copy/part stream 4
                    offsetValue: endian tmp
                    offsetValue = 0
            ]
    ]
    NumberOfPages: length? IFDOffsetList
]
```

The basic idea here is to store first the offset of each IFD in the file and the number of entries in a block! (IFDOffsetList) for a further access of each entry.

FD Entry

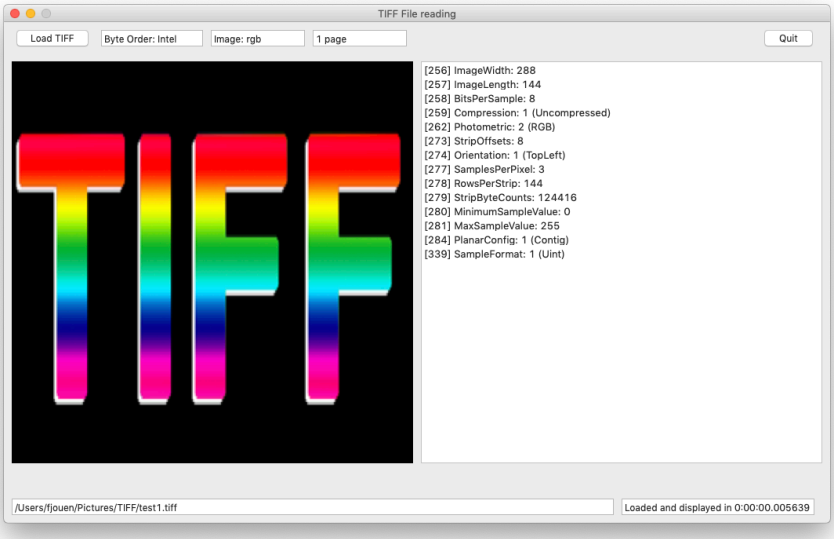
Each entry consists of 12 bytes. The first two bytes identifies the tag type (as in Tagged Image File Format). The next two bytes are the field type (byte, ASCII, short int, long int, ...). The next four bytes indicate the number of values. The last four bytes is either the value itself or an offset to the value. Values are expected to begin on a word boundary; the corresponding Value Offset will thus be an even number. This offset may point anywhere in the file, even after the image data. To save time and space the Value Offset contains the Value instead of pointing to the Value ***if and only if the Value fits into 4 bytes***.

Since we have an index offset for each IFD and the number of entries associated to the IFD, redCV can decode entries with the next function.

```
_readIFD: func [index [integer!]] [
            clear tagList
            bloc: IFDOffsetList/:index
            startOffset: first bloc
            numberOfEntries: second bloc
            i: 1
            while [i <= numberOfEntries ] [
                    tagOffset: startOffset +  (12 * (i - 1)) + 2
                    ; read directory entry (12 bytes)
                    tiff: head tiff
                    tiff:  skip tiff tagOffset
                    stream: copy/part tiff 12
                    tmp: copy/part stream 2
            TImgFDEntry/tiffTag: endian tmp ; get tag number
            stream: skip stream 2
            tmp: copy/part stream 2
            TImgFDEntry/tiffDataType: endian tmp ; tag type
            stream: skip stream 2
            tmp: copy/part stream 4
            TImgFDEntry/tiffDataLength: endian tmp; The number of values, Count of the indicated Type.
            stream: skip stream 4
            tmp: copy/part stream 4
            TImgFDEntry/tiffOffset: endian tmp ; value or offset data
            _getTagValue; get the tag value OK
                    i: i + 1
            ]
]
```

This function includes *_getTagValue* function in order to get the tag value and *_getTagValue* includes *_processTag* function in order to process once the tags in the file.

Now we can read and write Tiff files with *rcvLoadTiffImage* and *rcvSaveTiffImage* functions.

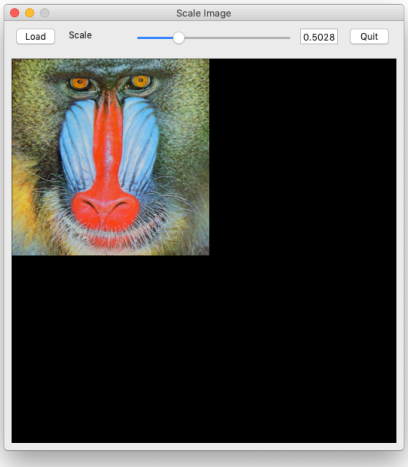


redCV/samples/ image_transformation

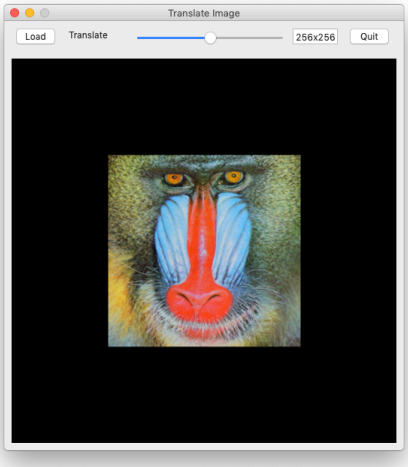
By default, Red includes very efficient spatial coordinates-based transformations that we can call with draw dialect. Let im[x, y] be an input image of size N × N. A spatial coordinates-based transformation, also called warping, aims at providing an image IM[k, l] from the input image im[x, y] according to IM[k, l] = im[x(k, l), y(k, l)]. x(k,l) and y(k,l) are the transformations or the pixel warping functions. These functions just modify **the spatial coordinates** of a pixel not its value.

imagescale.red illustrates the use of 2D linear transformations for scaling image. This is very useful for zooming image, for example. *rcvScaleImage* function uses Draw Dialect and you have to add the image instance to the draw block.

```
img1: rcvLoadImage %../../images/baboon.jpg
factor: 1.0
drawBlk: rcvScaleImage factor
append drawBlk [img1]
```



imageTranslate.red. The idea is very simple. A draw block is initialized with *rcvTranslateImage* function: drawBlk: rcvTranslateImage 0.50 0x0 image. Then coordinates are modified according to the slider value and draw block is updated with the new coordinates: factor: as-pair face/data * delta face/data * delta drawBlk/5: factor.

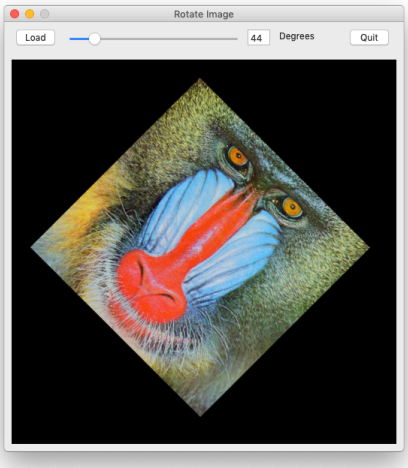


imageScroll.red. Similar approach is used for scrolling large images. Vertical and horizontal sliders update image position inside the canvas (a base! Red face):
xy/x: to integer! negate face/data * (max 0 img1/size/x - canvas/size/x)
xy/y: to integer! negate face/data * (max 0 img1/size/y - canvas/size/y)
Then draw block is updated: drawBlk/2: xy

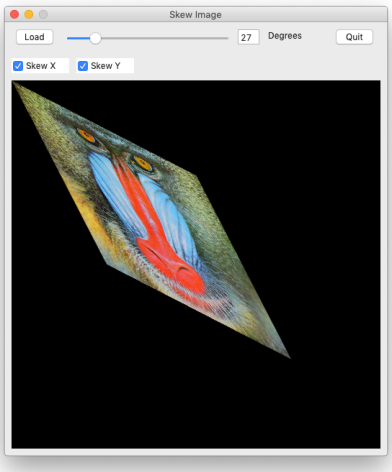
imageRotate.red. The rotation transformation is given by these equations:

$$x(k,l) = (k-x0)\cos\vartheta-(l-y0)\sin\vartheta+x0$$
$$y(k,l) = (k-x0)\sin\vartheta+(l-y0)\cos\vartheta+y0$$

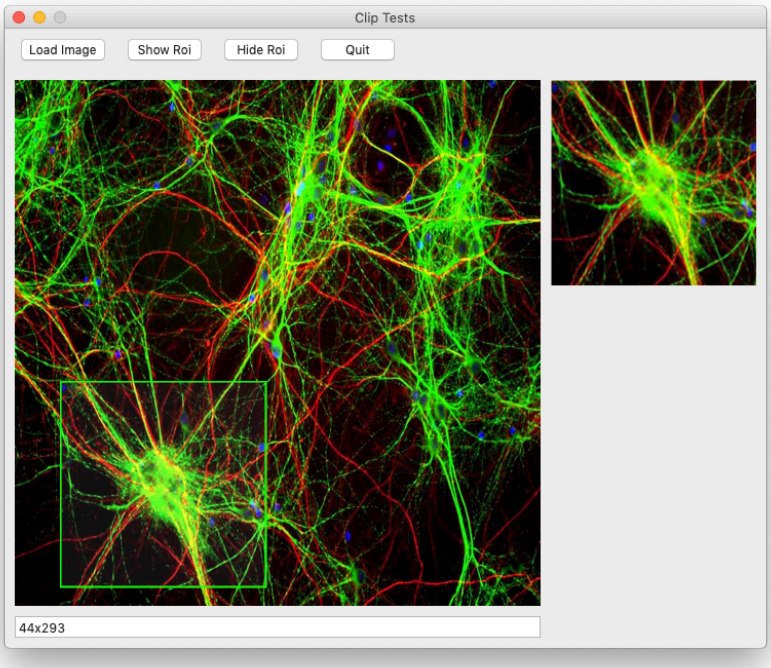
where [x0,y0] is the spatial coordinates of the center of the rotation (center of image) and θ the angle. Thanks to Red Team for implementing very efficient rotation algorithm.



imageSkew.red. With this code you can stretch your image in X, Y or both directions.



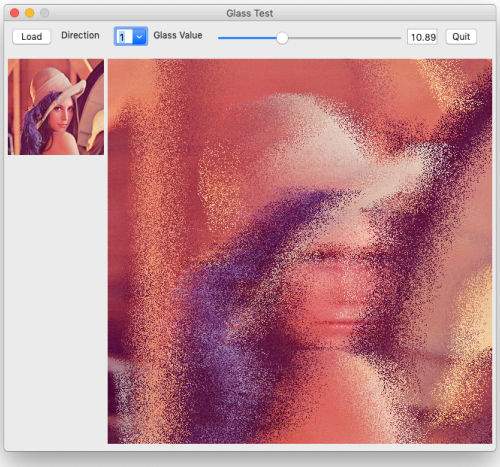
imageClip.red. Very often in image processing, we do not need to process the whole image, but only some parts of image called regions of interest (ROI). This can be done with *rcvClipImage* function and react method of Red faces.



imageFlip.red uses Red/System routine.
The vertical flip transformation is given by $x(i,j) = N-i$ and $y(i,j) = j$
The horizontal flip transformation is given by $x(I,i) = i$ and $y(I,i) = N-j$
where N is the image size, i the current column and j the current line of the image.

source Left/Right Up/Down Both

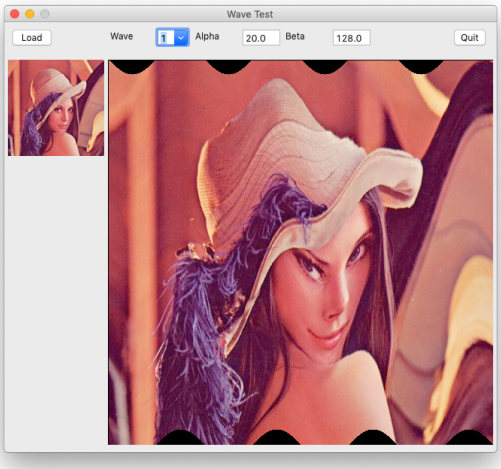
imageGlass.red. A glass effect is obtained by adding a small and random displacement (Glass value) to each pixel. With *rcvGlass* function, displacement can be vertical, horizontal or oblique. Enjoy.

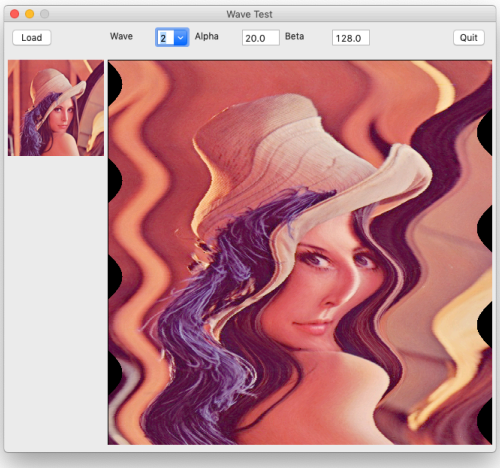


imageWave.red. *rcvWaveH* and *rcvWaveV* functions make a sinusoidal transform in Y or X axis according the equations:

$$y: y + (\alpha * \sin(2.0 * \pi * x / \beta))$$
$$x: x + (\alpha * \sin(2.0 * \pi * y / \beta))$$

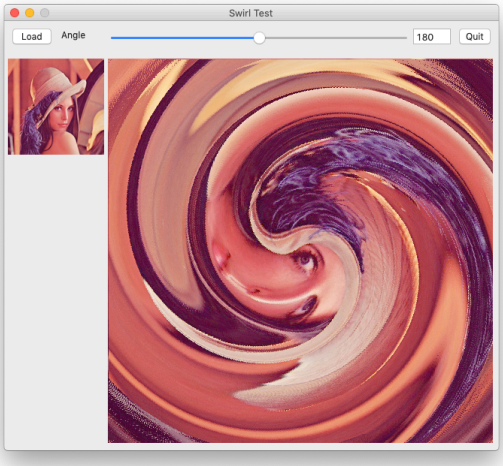
α and β are used to strengthen the funny effect.





imageSwirl.red. The swirl effect is a rotation transform, but the angle of the rotation θ varies with the pixel distance from the center of the image [x0,y0]:

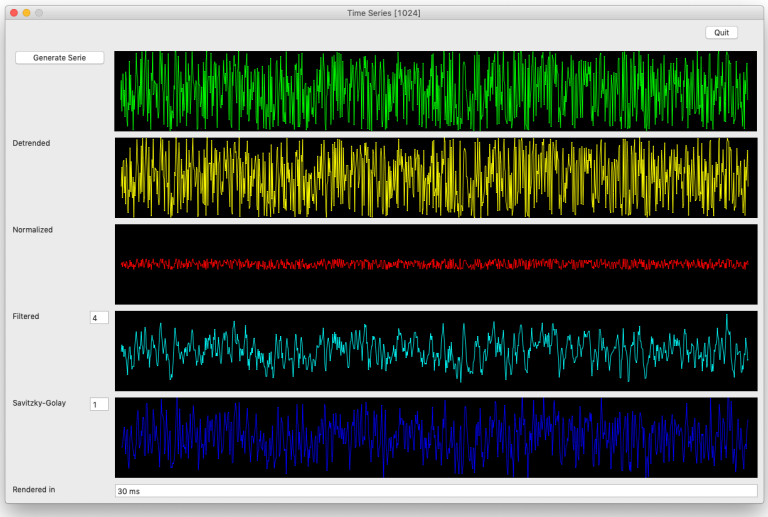
d: sqrt((x * x) + (y * y))
theta: pi / angle * d
x: (x * cos theta) - (y * sin theta) + x0
y: (x * sin theta) + (y * cos theta) + y0



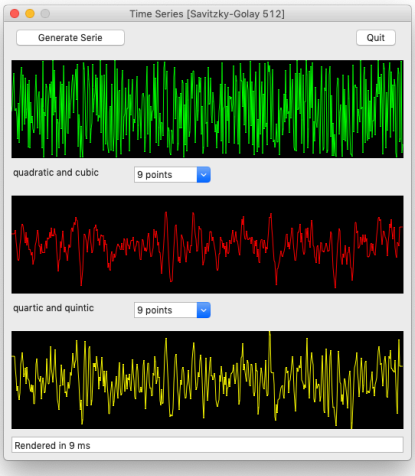
redCV/samples/ signal_processing

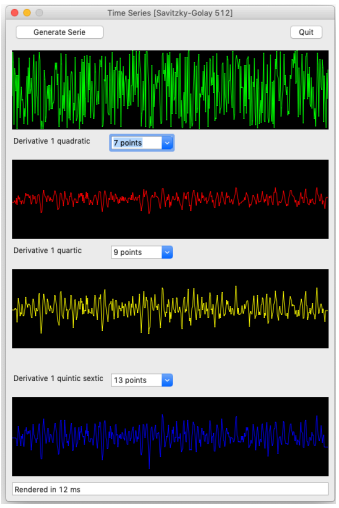
As previously seen in image_contours section, we sometimes need to use 1-D or 2-D signal processing algorithms to analyze the data contained in image.

timeSeries.red is a basic code for 1-D signal filtering and normalizing data.

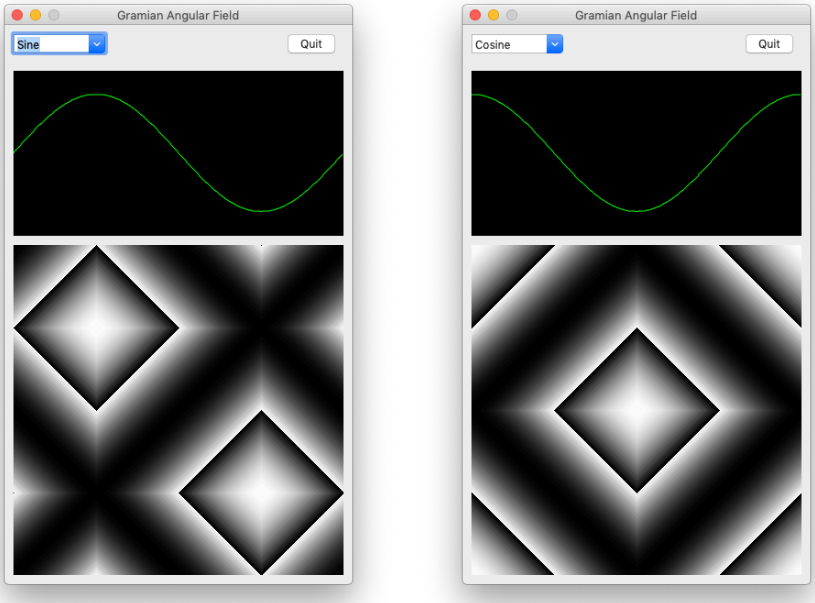


sgFilter.red and **sgDerivative.red** concern Savitzky–Golay filter. This filter belongs to digital filters classes that can be applied to a set of data points for smoothing the data without distorting the signal tendency. This is achieved by convolution and fitting successive sub-sets of adjacent data points with a low-degree polynomial by the method of linear least squares. Actually, the code is for 1-D signal and will be extended to 2-D signal such as images as soon as possible.





Gramian Angular Field (**gramian.red** and **gramian2.red**) is an image obtained from a time series, representing temporal correlation between each time point.



DTW

Fast Fourier Transform

redCV supports FFT. Thanks a lot, to Mel Cepstrum and Toomas Voglaid for their initial code on 1-D FFT. redCV FFT code is based on http://paulbourke.net/miscellaneous/ code.

http://cns-alumni.bu.edu/~slehar/fourier/fourier.html

redCV/samples/image_video

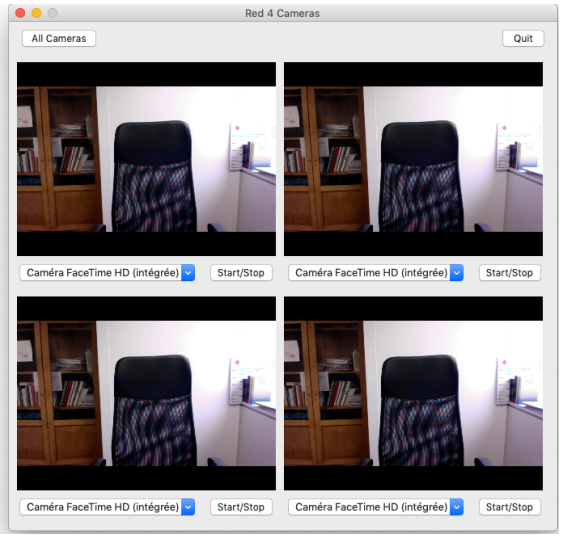
As known, support of video by Red is under progress. However, USB webcam can easily be accessed with Red due to the presence of the camera object. Thanks a lot, to Qingtian Xie for developing this object. Camera type is used to display a video camera feed. Camera/data facet lists the connected cameras as a block of strings and camera/selected facet selects the camera to display. Lastly, camera/image facet can be used to save the content of a camera. This last facet is the best way to capture the content of a camera is to use to-image function. This is illustrated in **cam1.red and cam2.red** samples.

```
Red [
        Title:   "Test camera Red VID "
        Author:  "Francois Jouen"
        File:     %camera.red
        Needs:    'View
]
;'
iSize: 320x240
margins: 10x10
cam: none; for camera object

view win: layout [
                title "Red Camera"
                origin margins space margins
                cam: camera 20x15; nonvisible camera just to get back image
                cb: check "recycle"
                f: field 100
                pad 35x0
                btnQuit: button "Quit" 50 on-click [quit]
                return
                canvas: base iSize black on-time [canvas/text: form now/time
                                        canvas/image: cam/image
                                        cam/image: none
                                        if cb/data [recycle]
                                        f/text: form stats
                                        ] font-color red font-size 12
                return
                cam-list: drop-list 220 on-create [face/data: cam/data]
                onoff: button "Start/Stop" on-click [
                                either cam/selected [
                                        cam/selected: none
                                        canvas/rate: none
                                        canvas/image: none
                                        canvas/text: ""
                                ][
                                        cam/selected: cam-list/selected
                                        canvas/rate: 0:0:0.04; max 1/25 fps in ms
                                        ]
                ]
                do [cam-list/selected: 1 canvas/rate: none
                        canvas/para: make para! [align: 'right v-align: 'bottom
                        cam/visible?: true]
                ]
]
```

Accessing multiple cameras

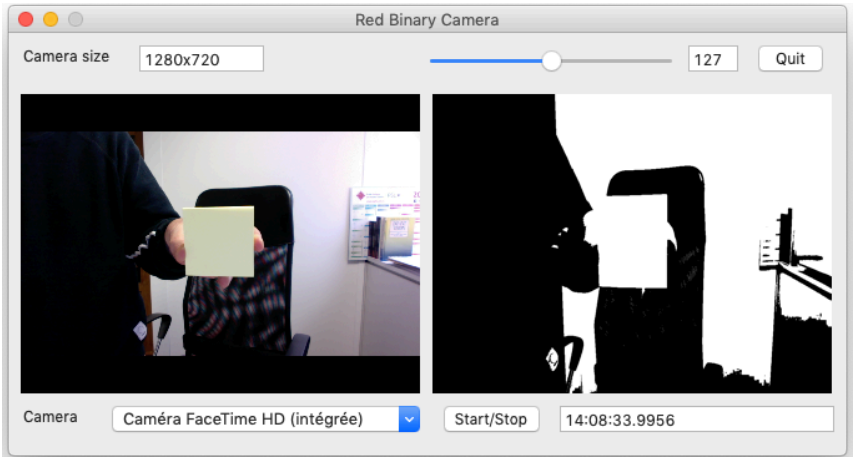
For experimental purposes, I often need to simultaneously record images from several cameras, and as explained in **cam4.red**, this is trivial with Red.



Here we use the same FaceTime camera but, you can use different USB cameras. This is really interesting for object 3-D reconstruction.

Of course, Red camera object can be associated with redCV routines and functions to create sophisticated video processing applications.

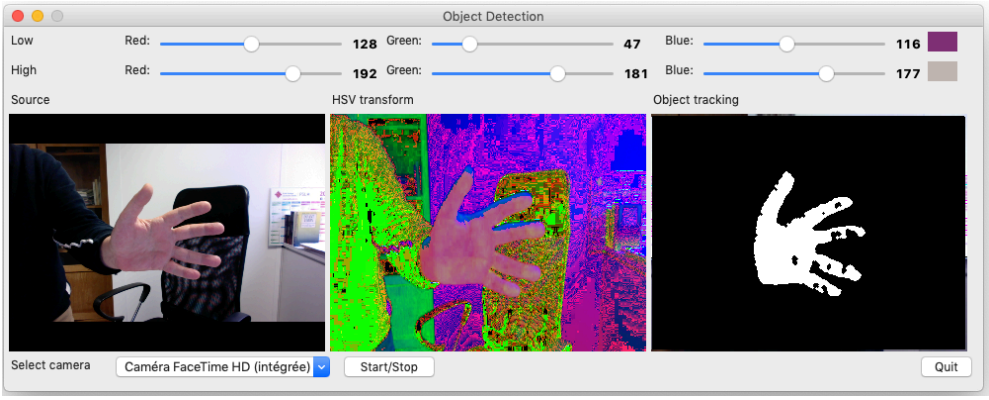
camConv.red and **camBin.red** give example of how to applicate simple filters on video in order to detect objects in video flow.



motion.red is an example of how to create a movement detector. Code is based on Collins, R., Lipton, A., Kanade, T., Fijiyoshi, H., Duggins, D., Tsin, Y., Tolliver, D., Enomoto, N., Hasegawa, O., Burt, P., Wixson, L.: A system for video surveillance and monitoring. Tech. rep., Carnegie Mellon University, Pittsburg, PA (2000). Basic idea is to compare successive images and to calculate the absolute difference between images. Then we apply a binary thresholding filter

in order to improve the precision of the detector. Result is simple. If there is no motion between successive images, resulting image is a black image. Otherwise, motion pixels appear in image as white pixels.

tracking.red and **tracking2.red** are a good illustration of object (here my hand) tracking in video. Code calls different redCV functions for RGB to HSV transform, and in range color selection. Then we use some morphogical operators in order to improve the tracking. This can be used for face identification.



Initially, my intention was to develop a specific red format for storing video images in a file for a further reading.

RCAM (RedCAMera) Format

RCAM files contain rgb values.
RCAM files are stored as binary values.
RCAM files contain a 32-byte header starting at offset 0 within the binary file. Video data are beginning just after the header at offset 32.
RgbSize (Image x size * Image y size * 3) is used to calculate the offset of each image contained in the file.

Offset	Size	Description
0	4	RCAM Four CC Red signature
4	4	Number of images in the file
8	4	Image x size
12	4	Image y size
16	8	duration in sec (float value)
24	4	Frames by Sec
28	4	compressed data (1) or not (0)
32	Image x size * Image y size * 3	Binary values for all images

However, this solution was memory consuming and I found a better way to do the job with ffmpeg. You will find here https://github.com/ldci/ffmpeg some tools for reading and writing video files from Red camera.