RedCV Open Source Computer Vision Library

What is RedCV

RedCV means Red Language Open Source Computer Vision Library. It is a collection of Red functions and routines that give access to many popular Image Processing algorithms.

The key features

RedCV provides cross-platform high level API that includes many functions. RedCV has no strict dependencies on external libraries. RedCV is free for both non-commercial and commercial use.

Who created it

The list of authors and major contributors:
François Jouen for the library development.
Thanks to Nénad Rakocevic and Qingtian Xie for developing Red and their constant help.
Thanks to Didier Cadieu for samples optimization.
Thanks to Bruno Anselme for ZLib binding.
Thanks to Fyodor Shchukin for illustration.

Where to get RedCV

Go https://github.com/ldci/redCV.

RedCV Reference Manual

- Using RedCV library
- Basic structures
- Images and matrices basic operators
- Image and matrix utilities
- Format conversion
- Color and color space conversion
- Arithmetic operators
- Logic operators
- Statistics and image features extraction
- Geometrical transformations
- Distances
- Dynamic Time Warping
- Image enhancement
- Thresholding
- Spatial Filtering
- Fast Edge Detection
- Lines detection
- Shapes detection
- Mathematical morphology
- GUI functions
- Random generator

Using RedCV Library

Most of functions are calling Red/System routines for faster image rendering. All redCV routines can be directly called from a red program (not for newbies). For a more convenient access, Red/System routines are exported as red functions. All red routines are prefixed with underscore (e.g. _rcvCopy). **Only red functions are documented**.

All includes to redCV libraries are declared in a single file (/libs/redcv.red). You just need including *redcv.red* file in your Red programs.

```
[
#include %core/rcvCore.red              ; Basic image creating and processing functions
#include %highgui/rcvHighGui.red        ; Fast Highgui functions
#include %matrix/rcvMatrix.red          ; Matrices functions
#include %imgproc/rcvImgProc.red        ; Image processing functions
#include %math/rcvRandom.red            ; Random laws for generating random images
#include %math/rcvStats.red             ; Statistical functions for images
#include %math/rcvDistance.red          ; Distance algorithm for detection in images
#include %ZLib/rcvZLib.red              ; ZLib compression algorithms
#include %tiff/rcvTiff.red              ; Tiff image reading and writing]
```

```
; all we need for computer vision with Red
#include %../../libs/redcv.red ; for red functions
```

Some lectures
Image Processing in C, by Dwayne Phillips. The first edition of Image Processing in C (Copyright 1994, ISBN 0-13-104548-2) was published by R & D Publications
1601 West 23rd Street, Suite 200
Lawrence, Kansas 66046-0127
Algorithms for Image Processing and Computer Vision (2011) by J.R. Parker, published by Wiley Publishing, Inc.
10475 Crosspoint Boulevard
Indianapolis, IN 46256

Basic Structures

Image

RedCV directly uses Red image! datatype. Loaded images by Red are in ARGB format (a tuple). Images are 8-bit and internally uses bytes [0..255] as a binary string. Images are 4-channels and actually Red can't create 1, 2 or 3-channels images. Similarly Red can't create 16-bit (0..65536) 32-bit or 64-bit (0.0..1.0) images.

Each pixel channel ARGB is represented by a byte! The byte! datatype's purpose is to represent unsigned integers in the 0-255 range. Many libraries use a byte pointer to access ARGB components of a pixel. Red proposes an optimized which uses an integer to store ARGB values in a single value. Since the memory size of an integer is 32 bits, is really easy to store 4 bytes (8-bit) value with an integer. Consequently, a int-ptr! will be used to access pixel value.

Now to access to ARGB values stored in the integer, Red applies right shift operator, both unsigned right shift: >>> and signed right shift: >>

```
a: pix1/value >>> 24                 ; byte 1 [0-255] Alpha (transparency) channel
r: pix1/value and 00FF0000h >> 16    ; byte 2 [0-255] Red channel
g: pix1/value and FF00h >> 8         ; byte 3 [0-255] Green channel
b: pix1/value and FFh                ; byte 4 [0-255] Blue channel
```

To write back pixel values, Red call signed left shift: << operator

pixD/value: (a << 24) OR (r << 16) OR (g << 8) OR b

Matrix

Matrix! Datatype is not yet implemented by Red. 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.

.

Images and matrices basic operators

rcvCreateImage

Creates and returns empty (black) image
rcvCreateImage: function [size [pair!] return: [image!]
size : image size width and height as a pair

```
dst: rcvCreateImage 512x512
```

rcvGetImageSize

Returns Image Size as a pair!
rcvGetImageSize: function [src [image!] return: [pair!]]

rcvGetImageFileSize

Gets Image File Size as a pair!
rcvGetImageFileSize: function [fileName [file!] return: [pair!]]

rcvCreateMat

Creates 2D matrix
rcvCreateMat: function [type [word!] bitSize [integer!] mSize [pair!] return: [vector!]]
type: name of accepted datatype: char! | integer!| float!
bitSize: 8 for char!, 8 | 16 | 32 for integer!, 32 | 64 for float!
mSize: matrix size as pair

```
msize: 128x128
mat1: rcvCreateMat 'integer! 8 msize
mat2: rcvCreateMat 'integer! 16 msize
mat3: rcvCreateMat 'integer! 32 msize
```

rcvLengthMat

Returns matrix length
rcvLengthMat: function [mat [vector!] return: [integer!]]

rcvMakeRangeMat

Creates an ordered matrix
rcvMakeRangeMat: function [a [number!] b [number!] step [number!] return: [vector!]]

```
rcvMakeRangeMat -5.0 5.0 0.25 -> [-5.0 -4.75 -4.5 -4.25 -4.0 -3.75 -3.5 -3.25 -3.0 -2.75 -2.5 -
2.25 -2.0 -1.75 -1.5 -1.25 -1.0 -0.75 -0.5 -0.25 0.0 0.25 0.5 0.75 1.0 1.25 1.5 1.75 2.0 2.25 2.5
2.75 3.0 3.25 3.5 3.75 4.0 4.25 4.5 4.75 5.0]
```

rcvMakeRangeMat 1 10 1 -> [1 2 3 4 5 6 7 8 9 10]

rcvMakeIndenticalMat

Creates a matrix with identical values

rcvMakeIndenticalMat: func [type [word!] bitSize [integer!] vSize [integer!] value [number!]return: [vector!]]

v: rcvMakeIndenticalMat 'Integer! 32 10 1 -> [1 1 1 1 1 1 1 1 1 1]
v: rcvMakeIndenticalMat 'Integer! 32 10 5 -> [5 5 5 5 5 5 5 5 5 5]
v: rcvMakeIndenticalMat 'Float! 64 10 0.25 ->[0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25 0.25]

rcvMakeBinaryMat

Makes a binary matrix [0..1]

rcvMakeBinaryMat: function [src [vector!] dst: [vector!]

Source matrix is 16 or 32-bit matrix [0..255]

rcvReleaseImage

Releases image data

rcvReleaseImage: function [src [image!]]
src : image to remove

rcvReleaseAllImages

Delete all images

rcvReleaseAllImages: function [list [block!]]

loaded images must be stored into a block! before releasing

rcvReleaseMat

Releases Matrix

rcvReleaseMat: function [mat [vector!]]
mat: matrix to be released

Release functions will be probably removed according to Red garbage collector development.

rcvLoadImage

Loads image from file

rcvLoadImage: function [fileName [file!] return: [image!] /grayscale]
filename: name of the file to load as a Red file datatype
/grayscale: loads image as grayscale

tmp: request-file
if not none? tmp [img1: rcvLoadImage tmp img2: rcvLoadImage /grayscale]

rcvLoadTiffImage

loads TIFF image

rcvLoadTiffImage: func [f [file!]]

f: name of the Tiff file to load

tmp: request-file
if not none? tmp [rcvLoadTiffImage tmp]

Attention: you need to call rcvTiff2RedImage function in order to display the image
canvas/image: rcvTiff2RedImage

Uncompressed bilevel, grayscale, palette-color images and RGB with samples per pixel up to 4 are supported.

Samples Per Pixel is usually 1 for bilevel, grayscale, and palette-color images.
Samples Per Pixel is usually 3 for RGB images

rcvReadTiffImageData

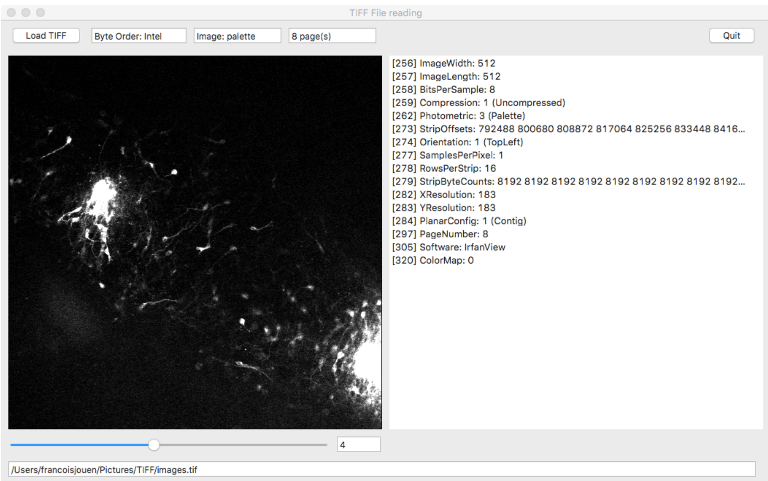
Reads multiple images included in TIFF File

rcvReadTiffImageData: func [page [integer!]]

page: number of image to access

Tiff files can include more than 1 image. You can use this function to access any image by its number.

Attention: you need to load first the tiff file before accessing image



rcvLoadImageAsBinary

Loads image from file and return image as binary

rcvLoadImageAsBinary: function [fileName [file!] return: [binary!] /alpha]
filename: name of the file to load as a Red file datatype
/ alpha: loads image as 4 channels image including alpha channel

rcvSaveImage

Save image to file

rcvSaveImage: function [src [image!] fileName [file!]]
src: image to save
filename: name of the file to save as a Red file datatype

Actually only png codec is supported for saving image. Will be improved in future by Red Team

rcvSaveTiffImage

Save red image as tiff

rcvSaveTiffImage: func [anImage [image!] f [file!] mode [integer!]]
anImage: image to save
f: name of the file
mode : 1 little endian (Intel) / 2 big endiand (Motorola)

rcvCloneImage

Returns a copy of source image

rcvCloneImage: function [src [image!] return: [image!]]
src: image to be cloned

```
img: recCreateImage 512x512
hsv: rcvCloneImage img
```

rcvCloneMat

Returns a copy of source matrix

rcvCloneMat: function [src [vector!] return: [vector!]
src: matrice to be cloned

rcvCopyImage

Copies source image to destination image

Source and destination image must have the same size!
rcvCopyImage: function [src [image!] dst [image!]]
src: image to be copied
dst: destination

```
img: hsv : recCreateImage 512x512
hsv: rcvCopy img hsv
```

rcvCopyMat

Copy source matrix to destination matrix

rcvCopyMat: function [src [vector!] dst [vector!]]
src: matrice to be copied
dst: destimation matrix

rcvZeroImage

Sets all image pixels to 0

rcvZeroImage: function [src [image!]]
src: image to clear

rcvRandomImage

Creates a random uniform color or pixel random image

rcvRandomImage: function [size [pair!] value [tuple!] /uniform /alea return: [image!]]
size: size of image as pair!
Value: random value as tuple!
/uniform : random uniform color
/alea : random pixels

rcvRandomMat

Randomizes matrix

rcvRandomMat: function [mat [vector!] value [integer!]]
mat: destinatiobn matrix
value: random value as integer!

```
mat1: rcvCreateMat 'integer! 8 msize
mat2: rcvCreateMat 'integer! 16 msize
mat3: rcvCreateMat 'integer! 32 msize
rcvRandomMat mat1 FFh
rcvRandomMat mat2 FFFFh
rcvRandomMat mat3 FFFFFFh
```

rcvColorImage

Set image color

function [src [image!] acolor [tuple!]]
src: image to color
acolor: required color as a tuple

rcvColorMat

Set matrix color

rcvColorMat: function [mat [vector!] value [integer!]]
mat: destination matrix
value: color value as integer

```
mat1: rcvCreateMat 'integer! 8 msize
rcvColorMat mat1 0
```

rcvSortMat

ascending sort of matrix

rcvFlipMat: function [v [vector!] return: [vector!]]]

rcvFlipMat

flip matrix

rcvFlipMat: function [v [vector!] return: [vector!]]

rcvCompressRGB

Compresses rgb image values

rcvCompressRGB: function [rgb [binary!] level [integer!] return: [binary!]]
rgb: rgb binary values of the image (image/rgb)
level: compression level for ZLib compression
[0: No compression
1: Best Speed
9: Best compression
-1: default compression]

rcvDecompressRGB

Uncompresses rgb image values

rcvDecompressRGB: function [rgb [binary!] bCount [integer!] return: [binary!]
rgb: previously rgb compressed values
bCount: size of non-compressed rgb values

```
rgb: copy img/rgb                        ; image rgb values
clevel: 9                                ; zLib best compression
result: copy #{}                         ; for compressed data
result2: copy #{}                        ; for uncompressed data
n: length? rgb                           ; size of uncompressed data
result: rcvCompressRGB rgb clevel        ; compress
result2: rcvDecompressRGB result n       ; uncompress
```

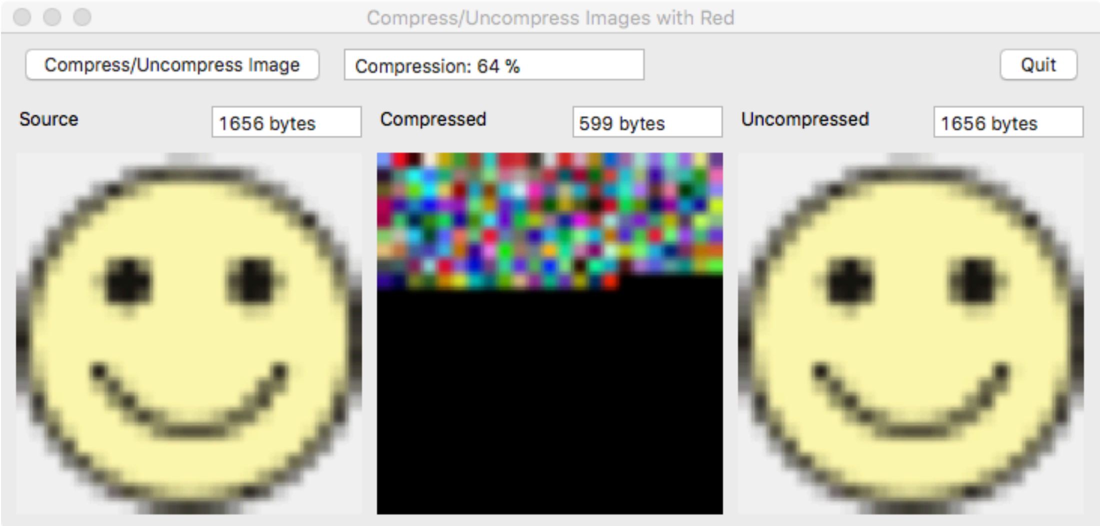

Compress/Uncompress Images with Red
Compress/Uncompress Image
Compression: 64 %
Quit
Source
1656 bytes
Compressed
599 bytes
Uncompressed
1656 bytes

Image and matrix utilities

rcvGetPixel

Returns pixel value at xy coordinates

rcvGetPixel: function [src [image!] coordinate [pair!] return: [tuple!]]
src: image
coordinate: xy position in image as a pair

rcvGetPixelAsInteger

Returns pixel value at xy coordinates as integer

rcvGetPixelAsInteger: function [src [image!] coordinate [pair!] return: [integer!]]
src: image
coordinate: xy position in image as a pair

rcvGetInt2D

Returns integer matrix value at xy coordinates

getInt2D: function [src [vector!] mSize [pair] coordinate [pair!] return: [integer!]]
src: source matrix
coordinate: xy position of the element as a pair
msize: matrix size as pair!

rcvGetReal2D

Returns float matrix value at xy coordinates

getInt2D: function [src [vector!] mSize [pair] coordinate [pair!] return: [integer!]]
src: source matrix
msize: matrix size as pair!
coordinate: xy position of the element as a pair

rcvSetPixel

Sets pixel value at xy coordinates

rcvSetPixel: function [src [image!] coordinate [pair!] val [tuple!]]
src: image
coordinate: xy position in image as a pair
val : pixel value as a tuple

rcvPokePixel

Set pixel value at xy coordinates

rcvPokePixel: function [src [image!] coordinate [pair!] val [tuple!]]

rcvSetInt2D

Sets value in integer matrix

setInt2D: function [dst [vector!] mSize [pair] coordinate [pair!] val [integer!]]
dst: destination matrix
msize: matrix size as pair!
coordinate: xy position of the element as a pair
val: value as integer

rcvSetReal2D

Sets value in float matrix
setInt2D: function [dst [vector!] mSize [pair] coordinate [pair!] val [float!]]
dst: destination matrix
msize: matrix size as pair
coordinate: xy position of the element as a pair
val: value as float

rcvMatleftPixel

Gets coordinates of first left pixel
rcvMatleftPixel: function [mat [vector!] matSize [pair!] value [integer!] return: [pair!]
mat: Integer matrix
matSize: matrix size as pair
value : pixel value (e.g. 1 or 255)

rcvMatRightPixel

Gets coordinates of first right pixel
rcvMatRightPixel: function [mat [vector!] matSize [pair!] value [integer!] return: [pair!]
mat: Integer matrix
matSize: matrix size as pair
value : pixel value (e.g. 1 or 255)

rcvMatUpPixel

Gets coordinates of first top pixel
rcvMatRightPixel: function [mat [vector!] matSize [pair!] value [integer!] return: [pair!]
mat: Integer matrix
matSize: matrix size as pair
value : pixel value (e.g. 1 or 255)

rcvMatDownPixel

Gets coordinates of first bottom pixel
rcvMatRightPixel: function [mat [vector!] matSize [pair!] value [integer!] return: [pair!]
mat: Integer matrix
matSize: matrix size as pair
value : pixel value (e.g. 1 or 255)

rcvSetAlpha

Sets image transparency

rcvSetAlpha: function [src [image!] dst [image!] alpha [integer!]
src: image remains unchanged and transparency is modified for destination image
alpha : transparency value [0..255]
sl: slider 256 [t: 255 - (to integer! sl/data * 255) rcvSetAlpha img1 img2 t]

rcvBlend

Computes the alpha blending of two images

rcvBlend: function [src1 [image!] src2 [image!] dst [image!] alpha [float!]]
src1: first image
src2: second image
dst: destination image
alpha: ratio of first image mixed with the second. Float value [0.0-1.0]

rcvBlendMat

Computes the alpha blending of two matrices

rcvBlendMat: function [mat1 [vector!] mat2 [vector!] dst [vector!] alpha [float!]]
mat1: first matrix
mat2: second matrix
dst: destination matrix
alpha: ratio of first matrix mixed with the second. Float value [0.0-1.0]

Format conversion

rcvImage2Mat

Converts Red Image to 8-bit 2-D Matrix

rcvImage2Mat: function [src [image!] mat [vector!]]
src: image
mat: vector
Grayscale

rcvMat2Image

Converts 8, 16 or 32-bit integer Matrix to Red Image

rcvMat82Image: function [mat [vector!] dst [image!]]
mat: vector
dst: image

rcvConvertMatScale

Converts Matrix Scale to another bit size

rcvConvertMatScale: function [src [vector!] dst [vector!] srcScale [number!] dstScale [number!] /fast /normal]
src: vector
dst: vector
srcScale: source range e.g 255
dstScale : destination range e.g 65535
/normal : uses a general function
/fast: uses a faster routine

```
msize: 256x256
mat1: rcvCreateMat 'integer! 8 msize
mat2: rcvCreateMat 'integer! 16 msize
mat3: rcvCreateMat 'integer! 32 msize
rcvConvertMatScale/normal mat1 mat2 FFh FFFFh
rcvConvertMatScale/normal mat1 mat3 FFh FFFFFFh
```

rcvMatInt2Float

Converts integer matrix to float [0..1] matrix

rcvMatInt2Float: function [src [vector!] dst [vector!] srcScale [float!]]
src: source matrix
dst: destination matrix
srcScale: source range as a float!

rcvMatFloat2Int

Converts float matrix to integer [0..255] matrix

rcvMatFloat2Int: function [src [vector!] dst [vector!]

src: source matrix

dst: destination matrix

rcvSplit

Separates source image in ARGB channels. Destination contains selected source channel.

rcvSplit: function [src [image!] dst [image!]/red /green /blue /alpha]

src: source image

dst: destination image

/red: red channel

/green: green channel

/blue: blue channel

/alpha: alpha channel

rcvSplit2Mat

Separates image channels to 4 8-bit matrices

rcvSplit2Mat: function [src [image!] mat0 [vector!] mat1 [vector!] mat2 [vector!] mat3 [vector!]]

src: image

mat0: image alpha channel

mat1: image red channel

mat2: image green channel

mat3: image blue channel

if source image is grayscale then mat1 = mat2 = mat3.

rcvMerge2Image

Merges 4 8-bit matrices to Red image

rcvMerge2Image: function [mat0 [vector!] mat1 [vector!] mat2 [vector!] mat3 [vector!] dst [image!]]

mat0: image alpha channel

mat1: image red channel

mat2: image green channel

mat3: image blue channel

dst: image

rcvTiff2RedImage

Converts TIFF image to Red image

rcvTiff2RedImage: func [return: [image!]]

Attention: you need a loaded Tiff File. See rcvLoadTiffImage documentation.

Color and color space conversion

rcvInvert

Destination image: inverted source image (Similar to NOT image)
rcvInvert: function [source [image!] dst [image!]]
src: source image
dst: destination image

rcv2BW

Converts RGB image to black[0] and white [255]
rcv2BW: function [src [image!] dst [image!]]
src: source image
dst: destination image

rcv2WB

Converts RGB image to white [255] and black[0]
rcv2BW: function [src [image!] dst [image!]]
src: source image
dst: destination image

rcv2BW: background = 0
rcv2WB: background = 255
Internal threshold value equal to 128. For an accurate thresholding see rcv2BWFilter function.

rcv2Gray

Converts RGB image to Grayscale according to refinement
rcv2Gray: function [src [image!] dst [image!] /average /luminosity /lightness return: [image!]]
src: source image
dst: destination image

The average method simply averages the values: (R + G + B) / 3.
The lightness method averages the most prominent and least prominent colors: (max(R, G, B) + min(R, G, B)) / 2.
The luminosity method is a more sophisticated version of the average method. It also averages the values, but it forms a weighted average to account for human perception. The formula for luminosity is 0.21 R + 0.72 G + 0.07 B.

rcv2BGRA

Converts RGBA to BGRA

rcv2BGRA: function [src [image!] dst [image!]]
src: source image
dst: destination image

rcv2RGBA

Converts BGRA to RGBA

rcv2RGBA: function [src [image!] dst [image!]]
src: source image
dst: destination image

rcvRGB2HSV

RBG color to HSV conversion

rcvRGB2HSV: function [src [image!] dst [image!]]
src: image
dst: image

rcvBGR2HSV

BGR color to HSV conversion

rcvBGR2HSV: function [src [image!] dst [image!]]
src: image
dst: image

The Hue/Saturation/Value model was created by A. R. Smith in 1978. The coordinate system is cylindrical. The hue value H runs from 0 to 360º. The saturation S is the degree of purity and is from 0 to 1. Purity is how much white is added to the color. S=1 makes the purest color (no white). Brightness V also ranges from 0 to 1, where 0 is the black. There is no transformation matrix for RGB or BGR to HSV conversion, but R, G and B are converted to floating-point format and scaled to fit 0..1 range.

rcvRGB2HLS

RBG color to HLS conversion

rcvRGB2HLS: function [src [image!] dst [image!]]
src: image
dst: image

rcvBGR2HLS

RBG color to HLS conversion

rcvBGR2HLS: function [src [image!] dst [image!]]
src: image
dst: image

Also a cylindrical coordinates system. There is no transformation matrix for RGB or BGR to HLS conversion, but R, G and B are converted to floating-point format and scaled to fit 0..1 range.

rcvRGB2YCrCb

RBG color to YCrCb conversion

rcvRGB2YCrCb: function [src [image!] dst [image!]]
src: image
dst: image

rcvBGR2YCrCb

BGR color to YCrCb conversion

rcvBGR2YCrCb: function [src [image!] dst [image!]]
src: image
dst: image

There is no transformation matrix.
Y <- 0.299*R + 0.587*G + 0.114*B
Cr <- (R-Y)*0.713 + delta
Cb <- (B-Y)*0.564 + delta

rcvRGB2XYZ

RGB to CIE XYZ color conversion

rcvRGB2XYZ: function [src [image!] dst [image!]]
src: image
dst: image

rcvBGR2XYZ

BGR to CIE XYZ color conversion

rcvBGR2XYZ: function [src [image!] dst [image!]]
src: image
dst: image

To transform from XYZ to RGB the matrix transform used is :

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{bmatrix} * \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

rcvRGB2Lab

RBG color to CIE L*a*b conversion

rcvRGB2Lab: function [src [image!] dst [image!]]
src: image
dst: image

rcvRGB2Lab

RBG color to CIE L*a*b conversion

rcvBGR2Lab: function [src [image!] dst [image!]]
src: image
dst: image

R, G and B are converted to floating-point format and scaled to fit 0..1 range. R, G and B are first converted to CIE XYZ before processing. On output 0≤L≤100, -127≤a≤127, -127≤b≤127. The values are then converted to 8-bit images: L <- L*255/100, a <- a + 128, b <- b + 128.

rcvRGB2Luv

RBG color to CIE L*u*v conversion

rcvRGB2Luv: function [src [image!] dst [image!]]
src: image
dst: image

rcvRGB2Luv

RBG color to CIE L*u*v conversion

rcvBGR2Luv: function [src [image!] dst [image!]]
src: image
dst: image

R, G and B are converted to floating-point format and scaled to fit 0..1 range. R, G and B are first converted to CIE XYZ before processing. On output 0≤L≤100, -134≤u≤220, -140≤v≤122. The values are then converted to 8-bit images: L <- L*255/100, u <- (u + 134)*255/354, v <- (v + 140)*255/256.

Arithmetic operators

rcvAdd

dst: src1 + src2

rcvAdd: function [src1 [image!] src2 [image!] dst [image!]]
src1: image
src2: image
dst: image

rcvAddMat

dst: src1 + src2

rcvAddMat: function [src1 [vector!] src2 [vector!] return: [vector!]]
src1: first matrix
src2: second matrix

rcvAddLIP

Destination image: image 1 + image 2 (Logarithmic Image Processing)

rcvAddLIP : function [src1 [image!] src2 [image!] dst [image!]
src1: image
src2: image
dst: image

Computes the addition of the two input images, according to the LIP model (Logarithmic Image Processing). The LIP image addition is defined as:
dest(x,y)= src1(x,y)+ src(x,y) – (src1(x,y)* src2(x,y)) / M
where M is the number of gray tones (256 for byte image)

rcvSub

dst: src1 - src2

rcvSub: function [src1 [image!] src2 [image!] dst [image!]]
src1: image
src2: image
dst: image

rcvSubMat

dst: src1 - src2

rcvSubMat: function [src1 [vector!] src2 [vector!] return: [vector!]]
src1: first matrix
src2: second matrix

rcvSubLIP

Destination image: image 1 - image 2 (Logarithmic Image Processing)
rcvSubLIP: function [src1 [image!] src2 [image!] dst [image!]]
src1: image
src2: image
dst: image

Computes the difference of the two input images, according to the LIP model (Logarithmic Image Processing). The LIP image addition is defined as:
dest(x,y) = M*(src1(x,y) – src2(x,y)) / (M – src2(x,y))
where M is the number of gray tones (256 for byte image)

rcvMul

dst: src1 * src2
rcvMul: function [src1 [image!] src2 [image!] dst [image!]]
src1: image
src2: image
dst: image

rcvMulMat

dst: src1 * src2
rcvMulMat: function [src1 [vector!] src2 [vector!] return: [vector!]]
src1: first matrix
src2: second matrix

rcvDiv

dst: src1 / src2
rcvDiv: function [src1 [image!] src2 [image!] dst [image!]]
src1: image
src2: image
dst: image

rcvDivMat

dst: src1 / src2
rcvDivMat: function [src1 [vector!] src2 [vector!] return: [vector!]]
src1: first matrix
src2: second matrix

rcvMod

dst: src1 // src2 (modulo)

rcvMod: function [src1 [image!] src2 [image!] dst [image!]]
src1: image
src2: image
dst: image

rcvRem

dst: src1 % src2 (remainder)

rcvRem: function [src1 [image!] src2 [image!] dst [image!]]
src1: image
src2: image
dst: image

rcvRemMat

dst: src1 % src2

rcvRemMat: function [src1 [vector!] src2 [vector!] return: [vector!]]
src1: first matrix
src2: second matrix

rcvAbsDiff

dst: absolute difference src1 src2

rcvAbsDiff: function [src1 [image!] src2 [image!] dst [image!]]
src1: image
src2: image
dst: image

rcvMIN

dst: minimum src1 src2

rcvMIN: function [src1 [image!] src2 [image!] dst [image!]]
src1: image
src2: image
dst: image

rcvMAX

dst: maximun src1 src2

rcvMax: function [src1 [image!] src2 [image!] dst [image!]]
src1: image
src2: image
dst: image

rcvLSH

Left shift image by value

rcvLSH: function [src [image!] dst [image!] val [integer!]]
src: image
dst: image
val: integer

rcvRSH

Right shift image by value

rcvLSH: function [src [image!] dst [image!] val [integer!]]
src: image
dst: image
val: integer

rcvPow

dst: src ^integer! or Float! Value

rcvPow: function [src [image!] dst [image!] val [number!]]
src: image
dst: image
val: integer or float

rcvSqr

Image square root

rcvSqr: function [src [image!] dst [image!] val [number!]]
src: image
dst: image
val: integer or float

rcvExp

TBD requires float images

rcvLog

TBD requires float images

rcvMeanImages

dst: (src1 + src2) /2

rcvMeanImages: function [src1 [image!] src2 [image!] dst [image!]]
src1: image
src2: image
dst: image

rcvMeanMats

Calculate mean values for 2 matrices

rcvMeanMats: function [src1 [vector!] src2 [vector!] return: [vector!]
src1: matrix 1
src2: matrix 2

rcvAddS

dst: src + integer! value

rcvAddS: function [src [image!] dst [image!] val [integer!]]
src: image
dst: image
val: integer

rcvAddSMat

src + value

rcvAddSMat: function [src [vector!] value [integer!]]
src: matrix
value: integer

rcvAddT

dst: src + tuple! value

rcvAddT: function [src [image!] dst [image!] val [tuple!]]

rcvSubS

dst: src - integer! value

rcvSubS: function [src [image!] dst [image!] val [integer!]]
src: image
dst: image
val: integer

rcvSubSMat

src - value

rcvSubSMat: function [src [vector!] value [integer!]]
src: matrix
value: integer

rcvSubT

dst: src - tuple! value

rcvSubT: function [src [image!] dst [image!] val [tuple!]]

rcvMulS

dst: src * integer! value

rcvMulS: function [src [image!] dst [image!] val [integer!]]
src: image
dst: image
val: integer

rcvMulSMat:

dst: src * integer! value

rcvMulSMat: function [src [vector!] value [integer!]
src: matrix
val: integer

rcvMulT

dst: src * tuple! value

rcvMulT: function [src [image!] dst [image!] val [tuple!]]

rcvDivS

dst: src / integer! value

rcvDivS: function [src [image!] dst [image!] val [integer!]]
src: image
dst: image
val: integer

rcvDivSMat

src / value

rcvDivSMat: function [src [vector!] value [integer!]]
src: matrix
value: integer

rcvDivT

dst: src / tuple! value

rcvDivT: function [src [image!] dst [image!] val [tuple!]]

rcvModS

dst: src // integer! Value (modulo)

rcvModS: function [src [image!] dst [image!] val [integer!]]
src: image
dst: image
val: integer

rcvModT

dst: src // tuple! Value (modulo)

rcvModT: function [src [image!] dst [image!] val [tuple!]]

rcvRemS

dst: src % integer! Value (remainder)

rcvRemS: function [src [image!] dst [image!] val [integer!]]
src: image
dst: image
val: integer

rcvRemT

dst: src % tuple! Value (remainder)

rcvRemT: function [src [image!] dst [image!] val [tuple!]]

rcvRemSMat

src % value (remainder)

rcvRemSMat: function [src [vector!] value [integer!]]
src: matrix
value: integer

Logic operators

rcvAND

dst: src1 AND src2

rcvAND: function [src1 [image!] src2 [image!] dst [image!]]
src1: first image
src2: second image
dst: src1 and src2

rcvANDMat

Returns source1 AND source2

rcvAndMat: function [src1 [vector!] src2 [vector!] return: [vector!]
src1: first matrice
src2: second matrice

rcvOR

dst: src1 OR src2

rcvOR: function [src1 [image!] src2 [image!] dst [image!]]
src1: first image
src2: second image
dst: src1 or src2

rcvORMat

Returns source1 OR source2

rcvORMat: function [src1 [vector!] src2 [vector!] return: [vector!]
src1: first matrice
src2: second matrice

rcvXOR

dst: src1 XOR src2

rcvXOR: function [src1 [image!] src2 [image!] dst [image!]]
src1: first image
src2: second image
dst: src1 xor src2

rcvXORMat

Returns source1 XOR source2

rcvORMat: function [src1 [vector!] src2 [vector!] return: [vector!]
src1: first matrice
src2: second matrice

rcvNAND

dst: src1 NAND src2

rcvNAND: function [src1 [image!] src2 [image!] dst [image!]]
src1: first image
src2: second image
dst: src1 nand src2

rcvNOR

dst: src1 NOR src2

rcvNOR: function [src1 [image!] src2 [image!] dst [image!]]
src1: first image
src2: second image
dst: src1 nor src2

rcvNXOR

dst: src1 NXOR src2

rcvNOR: function [src1 [image!] src2 [image!] dst [image!]]
src1: first image
src2: second image
dst: src1 nxor src2

rcvNOT

dst: src1 NOT src2

rcvNOR: function [src1 [image!] src2 [image!] dst [image!]]
src1: first image
src2: second image
dst: src1 not src2

rcvANDS

Tuple value is use to create a colored image which is ANDed to source image. Result is copied to destination

rcvANDS: function [src [image!] dst [image!] value [tuple!]
src: source image
dst: image
value: tuple!

```
rcvANDS img1 dst 255.0.0.0; dst: add red color to img1
```

rcvORS

Tuple value is use to create a colored image which is ORed to source image. Result is copied to destination

rcvORS: function [src [image!] dst [image!] value [tuple!]
src: source image
dst: image
value: tuple!

rcvXORS

Tuple value is use to create a colored image which is XORed to source image. Result is copied to destination

rcvXORS: function [src [image!] dst [image!] value [tuple!]
src: source image
dst: image
value: tuple!

rcvANDSMat

And integer value to all element in source matrix

rcvANDSMat: function [src [vector!] value [integer!]
src: matrice
value: integer!

rcvORSMat

OR integer value to all element in source matrix

rcvANDSMat: function [src [vector!] value [integer!]
src: matrice
value: integer!

rcvXORSMat

XOR integer value to all element in source matrix

rcvANDSMat: function [src [vector!] value [integer!]
src: matrice
value: integer!

Statistics and image features extraction

rcvCountNonZero

Returns number of non-zero values in image or matrix
rcvCountNonZero: function [arr [image! vector!] return: [integer!]]
arr: image or vector

rcvSum

Returns sum value of image or matrix as a block of rgb values
rcvSum: function [arr [image! vector!] return: [block!] /argb]
arr: image or vector
/argb: includes alpha channel

rcvSumMat

Returns matrix sum as a float value
rcvSumMat: function [mat [vector!] return: [float!]]

rcvMean

Returns mean value of image or matrix as a tuple of rgb values
rcvMean: function [arr [image! vector!] return: [tuple!] /argb]
arr: image or vector
/argb: includes alpha channel

rcvMeanMat

Returns matrix mean as float value
rcvMeanMat: function [mat [vector!] return: [float!]]

rcvSTD

Returns standard deviation value of image or matrix as a block of rgb values
rcvSTD: function [arr [image! vector!] return: [tuple!] /argb]
arr: image or vector
/argb: includes alpha channel

rcvMedian

Returns median value of image or matrix as a block of rgb values
rcvMedian: function [arr [image! vector!] return: [tuple!] /argb]
arr: image or vector
/argb: includes alpha channel

rcvProdMat

Return matrix product as a float value

rcvProdMat: function [mat [vector!] return: [float!]]

rcvMinValue

Returns minimal value of image or matrix as a block of rgb values

rcvMinValue: function [arr [image! vector!] return: [tuple!]]
arr: image or vector

rcvMaxValue

Returns maximum value of image or matrix as a block of rgb values

rcvMaxValue: function [arr [image! vector!] return: [tuple!]]
arr: image or vector

rcvMaxMat

Return maximum of matrix as a number

rcvMaxMat: function [mat [vector!] return: [number!]]

cvMinMat

Return minimum of matrix as a number
rcvMinMat: function [mat [vector!] return: [number!]]

rcvMinLoc

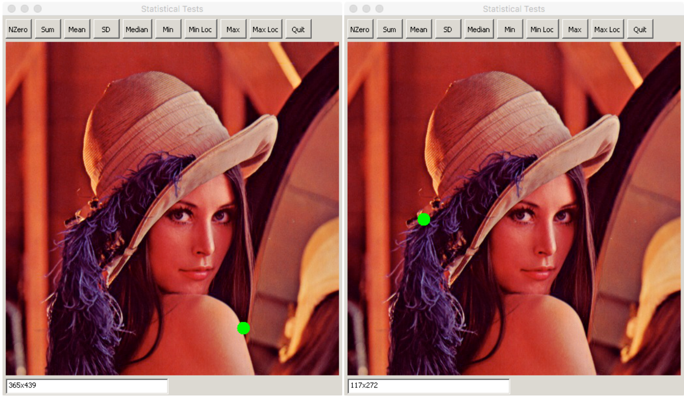
Finds global minimum location in array

rcvMinLoc: function [arr [image! vector!] arrSize [pair!] return: [pair!]]
arr: image or vector
arrSize: array size as pair

rcvMaxLoc

Finds global maximum location in array

rcvMaxLoc: function [arr [image! vector!] arrSize [pair!] return: [pair!]]
arr: image or vector
arrSize: array size as pair



Min Location Max Location

rcvHistogram

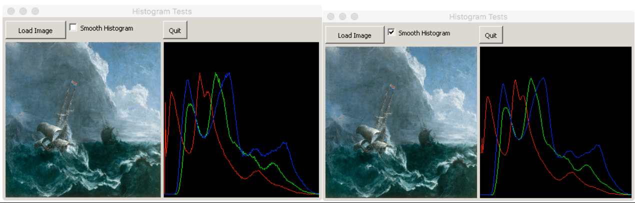
Calculates array histogram

rcvHistogram: function [arr [image! vector!] return: [vector!] /red /green /blue]
arr: image or vector
/red: histogram for red channel
/green: histogram for green channel
/blue: histogram for blue channel

rcvSmoothHistogram

This function smoothes the input histogram by a 3 points mean moving average

rcvSmoothHistogram: function [arr [vector!] return: [vector!]]
arr: input histogram as vector!



rcvRangeImage

Gives range value in image as a tuple

rcvRangeImage: function [source [image!] return: [tuple!]]
source: image

rcvGetMatSpatialMoment

Returns the spatial moment of the mat

```
rcvGetMatSpatialMoment: function [
	mat [vector!]
	matSize [pair!]
	p  [float!]
```

```
        q  [float!]
        return: [float!]
]
```

p - the order of the moment q - the repetition of the moment p: q: 0.0 -> moment order 0 -> form area

rcvGetMatCentralMoment

Returns the central moment of the mat

```
rcvGetMatCentralMoment: function [
        mat      [vector!]
        matSize [pair!]
        p          [float!]
        q          [float!]
        return: [float!]
]
```

rcvGetNormalizedCentralMoment

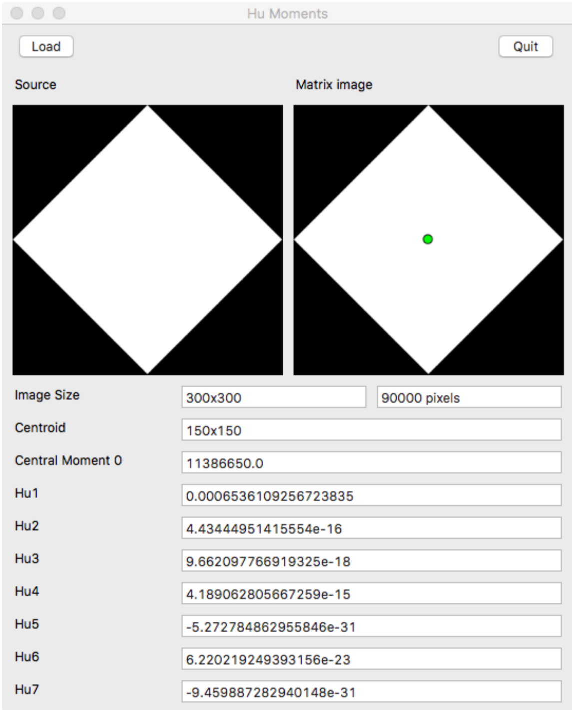
Return the scale invariant moment of the image

```
rcvGetNormalizedCentralMoment: function [
        mat      [vector!]
        matSize [pair!]
        p          [float!]
        q          [float!]
        return:  [float!]
]
```

rcvGetMatHuMoments

Returns Hu moments of the image

```
rcvGetMatHuMoments: function [
"Returns Hu moments of the image"
        mat      [vector!]
        matSize [pair!]
        return: [block!]
]
```



Hu Moments are normally extracted from the outline of an object in an image. An **image moment** is a particular weighted average of the image pixels' intensities, or a function of such moments, usually chosen to have some attractive property.

rcvSortImage

Ascending image sorting

rcvSortImage: function [source [image!] dst [image!]]

source: image

dst: image

rcvIntegral

Calculates integral images

rcvIntegral: function [src [image!] sum [image!] sqsum [image!]

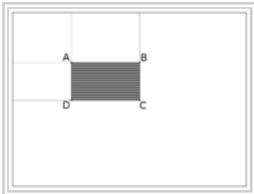
src: image or integer matrice

sum: image or integer matrice for summed area table

sqsum: image or integer matrice for square summed area table

mSize : image or integer matrice size as a pair!

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 etc.

Integral image *ii* is defined according to :

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

rcvQuickHull

Finds the convex hull of a point set

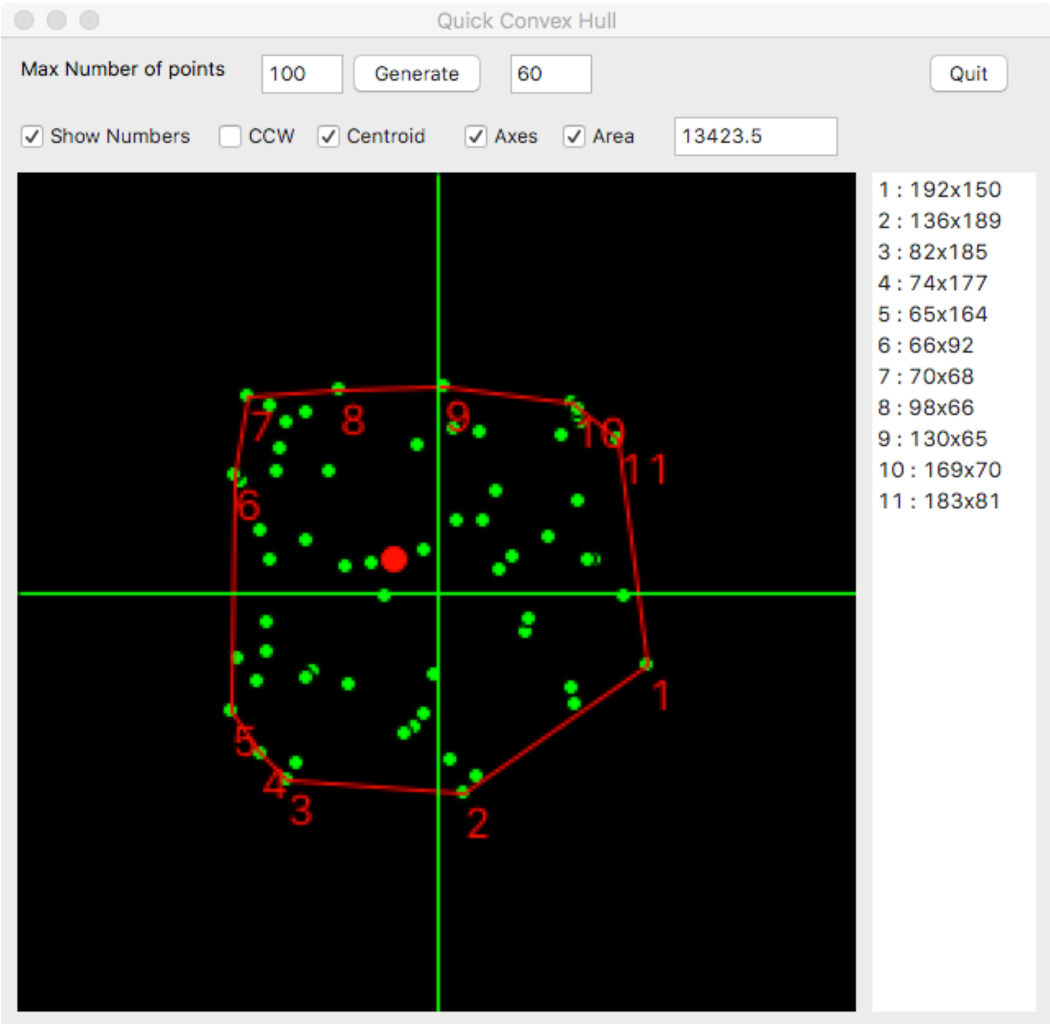
rcvQuickHull: function [points [block!] return: [block!] /cw/ccw]

points: Input 2D point set as pair!

Returns output convex hull as a block of pair

/cw/ccw : Orientation flag. If cw, the output convex hull is oriented clockwise. Otherwise, it is oriented counter-clockwise. The assumed coordinate system has its X axis pointing to the right, and its Y axis pointing upwards.

The convex hull problem in geometry tries to find the smallest convex set containing the points.



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.

rcvContourArea

Calculates the area of polygon generated by rcvQuickHull function

rcvContourArea: function [hull [block!] return: [float!] /signed]

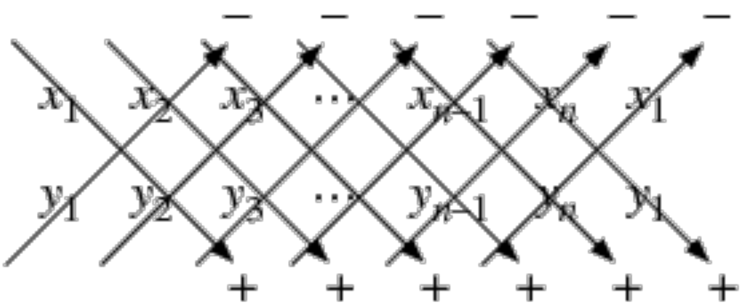
hull: list of coordinates (as pair!) generated by the rcvQuickHull function

Return: area as float!

If signed refinement is used returns the signed area

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:

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



See Weisstein, Eric W. "Polygon Area." From MathWorld--A Wolfram Web Resource. http://mathworld.wolfram.com/PolygonArea.html

Geometrical transformations

rcvFlip

Left/Right, Up/Down or both directions image flip

rcvFlip: function [src [image!] dst [image!] /horizontal /vertical /both return: [image!]]
src: source image
dst: destination image
refinement for direction

rcvResizeImage

Resizes image and applies filter for Gaussian pyramidal up or downsizing if required

rcvResizeImage: function [src [image!] iSize [pair!]/Gaussian return: [pair!]]
src: destination image
canvas: Red base face containing the image
iSize: New size of the image as pair
/Gaussian: applies a 5x5 kernel Gaussian filter on image

```
img1: rcvLoadImage %../../images/lena.jpg
dst:  rcvCreateImage img1/size
iSize: 256x256
canvas: base iSize dst
nSize: 512x512
rcvResizeImage dst nSize
canvas/size: nSize
dst: to-image canvas
```

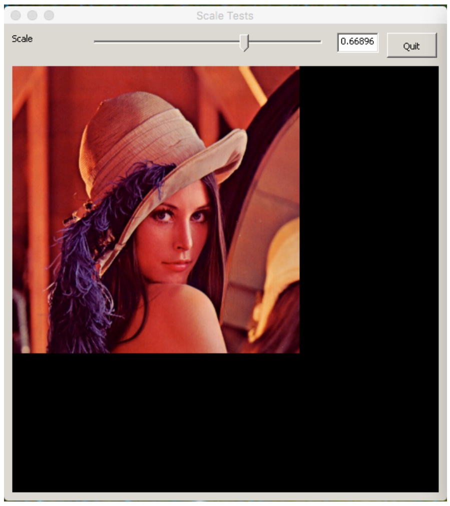
rcvScaleImage

Sets the scale factors: Returns a Draw block

rcvScaleImage: function [factor [float!] return: [block!]]

factor: scale factor as float! Default value : 1.0 original size

```
This function uses Draw Dialect and you have to add
the image instance to the draw block.
img1: rcvLoadImage %../../images/lena.jpg
factor: 1.0
drawBlk: rcvScaleImage factor
append drawBlk [img1]
…
```



rcvTranslateImage

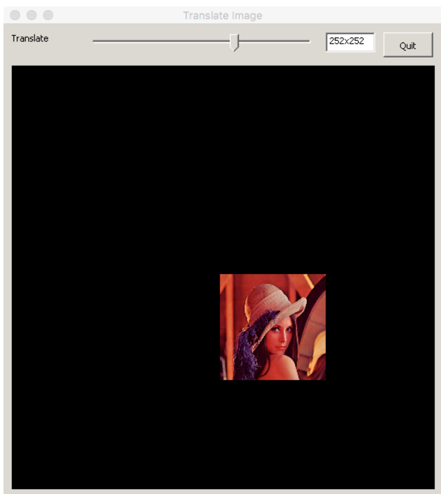
Sets the origin for drawing commands : Returns a Draw block

rcvTranslateImage: function [scaleValue [float!] translateValue [pair!] return: [block!]]

scaleValue: float! value to reduce or increase image size

translateValue : pair to translate image in X and Y direction

```
This function uses Draw Dialect and you have
to add the image instance to the draw block.
img1: rcvLoadImage %../../images/lena.jpg
factor: 0x0
drawBlk: rcvTranslateImage 0.25 factor
append drawBlk [img1]
…
```



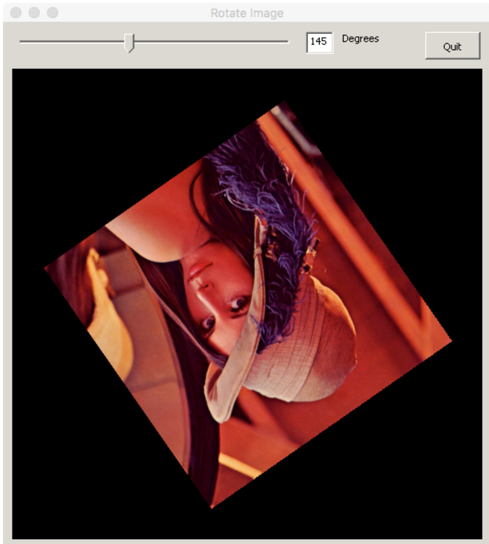
rcvRotateImage

Sets the clockwise rotation about a given point, in degrees : Returns a Draw block

rcvRotateImage: function [scaleValue [float!] translateValue [pair!] angle [float!] center [pair!] return: [block!]]

scaleValue: float! value to reduce or increase image size
translateValue : pair to translate image in X and Y direction
angle: rotation of image in degrees
center: center of image rotation as pair! Default value 0x0

```
This function uses Draw Dialect and you
have to add the image instance to the
draw block.
img1: rcvLoadImage %../../images/lena.jpg
iSize: img1/size
centerXY: iSize / 2
rot: 0.0
drawBlk: rcvRotateImage 0.625 96x96 rot
centerXY
append drawBlk [img1]
…
```



rcvSkewImage

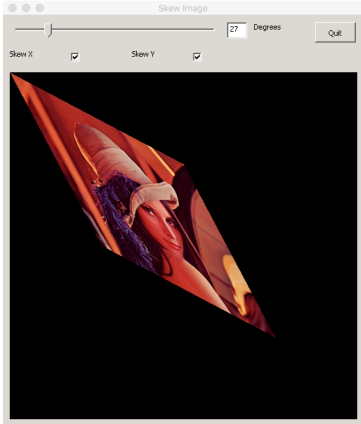
Sets a coordinate system skewed from the original by the given number of degrees

rcvSkewImage: function [scaleValue [float!] translateValue [pair!] x [number!] y [number!] return: [block!]]

scaleValue: float! value to reduce or increase image size
translateValue : pair to translate image in X and Y direction
x: skew along the x-axis in degrees (integer! float!).
y: skew along the y-axis in degrees (integer! float!).

This function uses Draw Dialect and you have to add the image instance to the draw block.

```
img1: rcvLoadImage %../../images/lena.jpg
x: 0
y: 0
drawBlk: rcvSkewImage 0.5 0x0 x y
append drawBlk [img1]
```



Chamfer Distances

rcvMakeGradient

Makes a gradient matrix for contour detection (similar to Sobel) and returns max gradient value

rcvMakeGradient: function [src [vector!] dst [vector!] mSize [pair!] return: [integer!]]
src: integer matrix
dst: integer matrix
mSize: matrix size as a pair!

rcvMakeBinaryGradient

Makes a binary [0 1] matrix for contour detection

rcvMakeBinaryGradient: function [src [vector!] mat [vector!] maxG [integer!] threshold [integer!]]
src: integer matrix
mat: destination integer matrix
maxG : max gradient value
threshold : integer value for binary thresholding

rcvFlowMat

Calculates the distance map to binarized gradient

rcvFlowMat: function [input [vector!] output[vector!] scale [float!] return: [float!]]
input: float source matrix
output: destination matrix including distances
returns max distance

rcvnormalizeFlow

Normalizes distance into 0..255 range according to scale value

rcvnormalizeFlow: function [input [vector!] factor [float!]]
input: integer matrix
factor: value used for normalization such as max gradient value

rcvGradient&Flow

Creates an image including flow and gradient values

rcvGradient&Flow: function [input1 [vector!] input2 [vector!] dst [image!]]
input1: flow integer matrix
input2: gradient integer matrix
dst: red image for mixing flow and gradient

rcvChamferDistance

Selects a pre-defined chamfer kernel

rcvChamferDistance: function [chamferMask [block!] return: [block!]]
Kernels calculated by Verwer, Borgefors and Thiel
cheessboard: copy [1 0 1 1 1 1]
chamfer3: copy [1 0 3 1 1 4]
chamfer5: copy [1 0 5 1 1 7 2 1 11]
chamfer7: copy [1 0 14 1 1 20 2 1 31 3 1 44]
chamfer13: copy [1 0 68 1 1 96 2 1 152 3 1 215 3 2 245 4 1 280 4 3 340 5 1 346 6 1 413]

rcvChamferCreateOutput

Creates a distance map (float!)
rcvChamferCreateOutput: function [mSize [pair!] return: [vector!]]
mSize: matrix size as a pair value

rcvChamferInitMap

Initializes distance map
rcvChamferInitMap: function [input [vector!] output [vector!]]
input: a binary [0/1] matrix
output: must be a vector of float!

If input value= 0, the point belongs to the object and thus the distance is 0.0
If input value= 1, the point is outside and the distance (-1.0) must be calculated

rcvChamferCompute

Calculates the distance map to binarized gradient
rcvChamferCompute: function [output [vector!] chamfer [block!] mSize [pair!]]
output: float matrix created by rcvChamferInitMap function is used
chamfer: selected pre-defined kernel used for distance calculation (e.g. chamfer5)

rcvChamferNormalize

Normalizes distance map
rcvChamferNormalize: function [output [vector!] value [integer!]]

Dynamic Time Warping

Quoting wikipedia:
"In time series analysis, dynamic time warping (DTW) is an algorithm for measuring similarity between two temporal sequences which may vary in time or speed. For instance, similarities in walking patterns could be detected using DTW, even if one person was walking faster than the other, or if there were accelerations and decelerations during the course of an observation."
Applied to computer vision DTW is really useful if we want to compare shapes and decide if shapes are similar or not.
In redCV we use a basic DTW algorithm which is documented here : https://nipunbatra.github.io/blog/2014/dtw.html. Thanks to Nipun Batra for writing a clear python code.
The objective is to find a mapping between all points of x and y series. In the first step, we will find out the distance between all pair of points in the two signals. Then, in order to create a mapping between the two signals, we need to create a path. The path should start at (0,0) and want to reach (M,N) where (M, N) are the lengths of the two signals. To do this, we thus build a matrix similar to the distance matrix. This matrix would contain the minimum distances to reach a specific point when starting from (0,0). DTW value corresponds to (M,N) sum value.

rcvDTWMin

Minimal value between 3 values
rcvDTWMin: function [x [number!] y [number!] z [number!] return: [number!]]

rcvDTWDistances

Making a 2d matrix to compute distances between all pairs of x and y series
rcvDTWDistances: function [x [block!] y [block!] return: [block!]]
x: a block! of number!
y: a block! of number!
Returns the distance matrix

rcvDTWRun

Making a 2d matrix to compute minimal distance cost
rcvDTWRun: function [x [block!] y [block!] dMat [block!] return: [block!]
x: a block! of number!
y: a block! of number!
Returns the minimal distance matrix

rcvDTWGetDTW

Returns DTW value
rcvDTWGetDTW: function [cMat [block!] return: [number!]] [
cMat: Minilam distance matrix
returns DTW value

rcvDTWGetPath

Finds the path minimizing the distance

rcvDTWGetPath: function [x [block!] y [block!] cMat [block!] return: [block!]

x: a block! of number!

y: a block! of number!

Returns optimal distance path

rcvDTWCompute

Short-cut to get DTW value if you don't need distance and cost matrices

rcvDTWCompute: function [x [block!] y [block!] return: [number!]

x: a block! of number!

y: a block! of number!

Returns DTW value

Code sample

```
dMatrix: rcvDTWDistances x y
cMatrix: rcvDTWRun x y dMatrix
dtw: rcvDTWGetDTW cMatrix
xPath: rcvDTWGetPath x y cMatrix
```

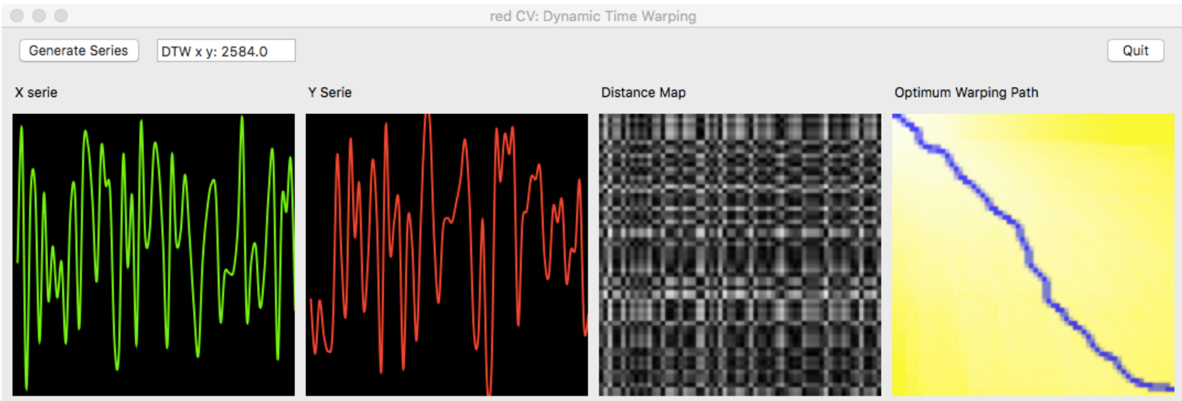


Image enhancement

rcvMakeTranscodageTable

Creates a transcoding 256 table for affine enhancement

rcvMakeTranscodageTable: function [n [percent!] return: [vector!]]

n: percent of values to exclude

This function is used by rcvContrastAffine method. See below.

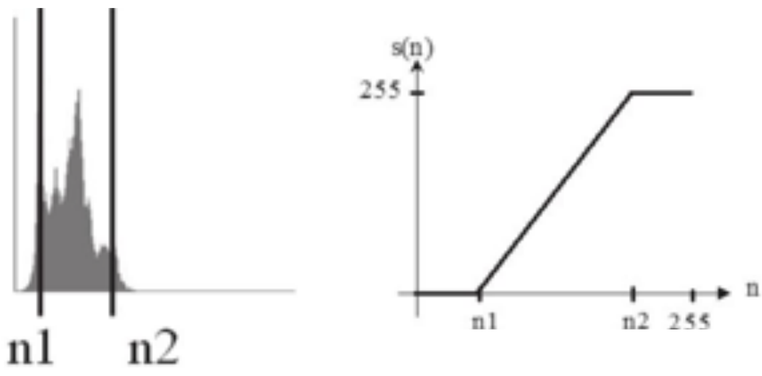
rcvContrastAffine

Enhances image contrast with affine function

rcvContrastAffine: function [image [vector!] n [percent!]]

image: a 8-bit matrice

n: percent of values to exclude



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

rcvHistogramEqualization

This function performs histogram equalization on the input image array

rcvHistogramEqualization: function [image [vector!]   gLevels [integer!]]

image: a 8-bit matrice

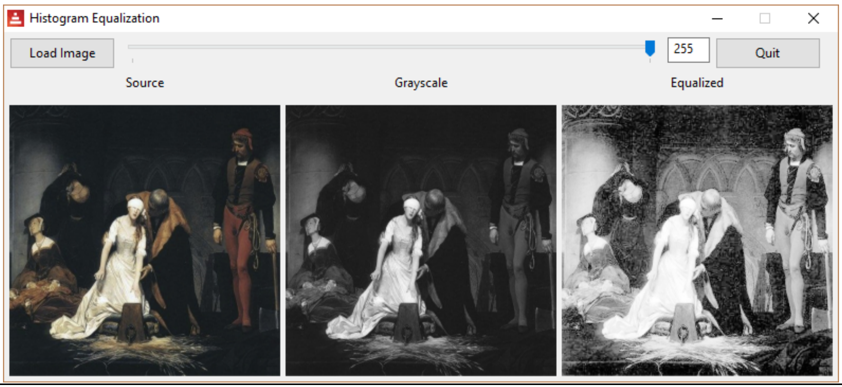
gLevels: number of gray levels in the new image

This algorithm performs first the histogram of the source image and then calculates the probability-density function of a pixel value n: $p_1(n)$ is the probability of finding a pixel with the value n in the image. Then the following equation is applied

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

$H_c(a)$ is the histogram of the original image c and D_m 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 c.

This function is useful for improving contrast in low-contrasted images or simply modifying the contrast of image.



Thresholding

rcv2BWFilter

Binarization of RGB image according to threshold value

rcv2BWFilter: function [src [image!] dst [image!] thresh [integer!]]
src: image
dst: image
thresh: threshold integer value

rcvThreshold

Applies fixed-level threshold to array elements. Images are processed as grayscale.

rcvThreshold: function [src [image!] dst [image!] thresh [integer!] mValue [integer!] /binary /binaryInv /trunc /toZero /toZeroInv
src: image
dst: image
thresh: threshold integer value
mValue: maximal integer value
refinements are used for thresholding type

binary:dst(x,y) = mValue, if src(x,y)>threshold, 0, otherwise
binarryInv: dst(x,y) = 0, if src(x,y)>threshold, mVvalue, otherwise
trunc: dst(x,y) = threshold, if src(x,y)>threshold , src(x,y), otherwise
toZero: dst(x,y) = src(x,y), if src(x,y)>threshold, 0, otherwise
toZeroInv: dst(x,y) = 0, if src(x,y)>threshold, src(x,y), otherwise

rcvInRange

Extracts sub array from image according to lower and upper rgb values

rcvInRange: function [src [image!] dst [image!] lower [tuple!] upper [tuple!] op [integer!]]
src: source image
dst: destination image
lower: lower tuple
upper: lower tuple
op: if op = 0 image is binarized else colors are extracted

rcvInRangeMat

Extracts sub array from matrix according to lower and upper values

rcvInRangeMat: function [src [vector!] dst [vector!] lower [integer!] upper [integer!] op [integer!]
src: source matrix
dst: destination matrix
lower: lower value
upper: lower upper
op: if op = 0 image is binarized else gray values are extracted

Spatial filtering

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

rcvMakeGaussian

Creates a Gaussian uneven kernel

rcvMakeGaussian: function [kSize [pair!] return: [block!]]
kSize: uneven pair for kernel e.g 3x3

Creates a Gaussian uneven kernel with 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.

rcvGaussianFilter

Fast Gaussian 2D filter

rcvGaussianFilter: function [src [image!] dst [image!]]
src: image
dst: image

Kernel is 3x3 and bias is equal to zero. For larger kernel please use rcvFilter2D.
knl: rcvMakeGaussian 11x11 rcvFilter2D src dst

rcvConvolve

Convolves an image with the kernel

rcvConvolve: function [src [image!] dst [image!] kernel [block!] factor [float!] delta [float!]]
src: source image
dst: destination image
kernel: kernel matrix as block!
factor: multiplier factor as float!

delta: bias for image brightness

```
img1: rcvLoadImage %../../images/lena.jpg
dst:  rcvCreateImage img1/size
gaussian: [0.0 0.2 0.0
           0.2 0.2 0.2
          0.0 0.2 0.0]
rcvConvolve img1 dst gaussian 2.0 0.0
```

rcvConvolveMat

Convolves a 2-D matrix with the kernel

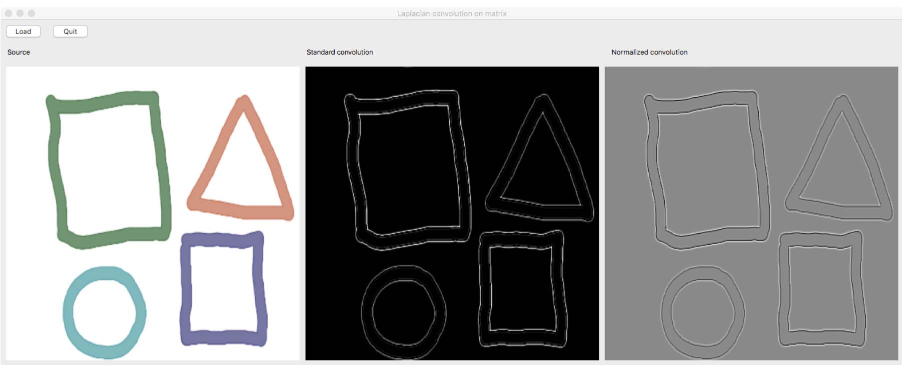
rcvConvolveMat: function [src [vector!] dst [vector!] mSize[pair!] kernel [block!] factor [float!] delta [float!]]
src: source matrix
dst: destination matrix
mSize: matrix size as a pair!
kernel: kernel matrix as block!
factor: multiplier factor as float!
delta: bias for image brightness

rcvConvolveNormalizedMat

Convolves a 2-D matrix with the kernel and applies a scale to result

rcvConvolveNormalizedMat: function [src [vector!] dst [vector!] mSize[pair!] kernel [block!] factor [float!] delta [float!]]
src: source matrix
dst: destination matrix
mSize: matrix size as a pair!
kernel: kernel matrix as block!
factor: multiplier factor as float!
delta: bias for image brightness

This function is two-pass: First, calculates minimal and maximal weighted sums resulting from the convolution process. This allows to calculate a scale equal to 255 / (maximal – minimal). Then each matrix convoluted value is rescaled by (value – minimal) * scale. This means that whatever the sign of convoluted values, values are transformed into bytes [0..255] values.



rcvFastConvolve

Convolves 8-bit and 1-channel image with the kernel

rcvFastConvolve: function [src [image!] dst [image!] channel [integer!] kernel [block!] factor [float!] delta [float!]]

src: source image

dst: destination image

channel: image channel to process (RGB)

kernel: kernel matrix as block!

factor: multiplier factor as float!

delta: bias for image brightness

rcvFilter2D

Basic convolution filter

rcvFilter2D: function [src [image!] dst [image!] kernel [block!] delta [integer!]]

src: image

dst: image

kernel: kernel matrix as block!

delta: bias for image brightness

Similar to convolution but the sum of the weights is computed during the summation, and used to scale the result.

rcvFastFilter2D

Fast convolution filter

rcvFastFilter2D: function [src [image!] dst [image!] kernel [block!]]

src: image

dst: image

kernel: kernel matrix as block!

A faster version without controls on pixel value! Basically for 1 channel gray scaled image. The sum of the weights is computed during the summation, and used to scale the result

rcvPointDetector

Convolution allowing to find dots in image or matrix

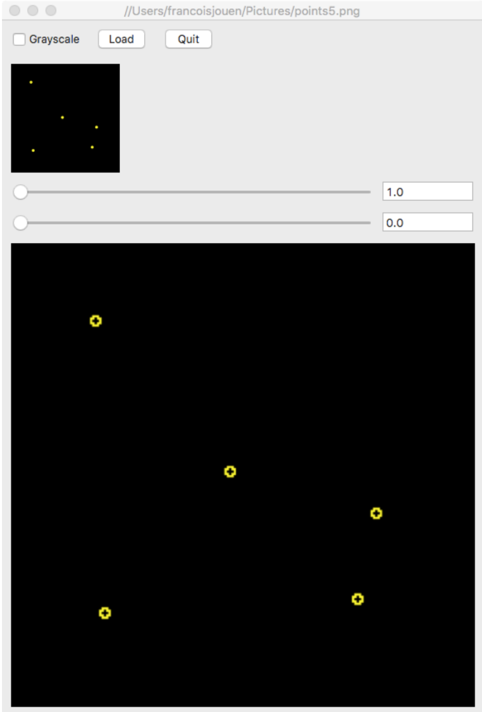
rcvPointDetector: function [src [image! vector!] dst [image! vector!] param1 [float!] param2 [float!]]

src: source image or matrix

dst: destination image or matrix

param1 : threshold value

param2 : luminance value



//Users/francoisjouen/Pictures/points5.png
Grayscale
Load
Quit
1.0
0.0

Fast edge detection

You can, of course, build your own edges detectors by convolution (see rcvConvolve function). Here are included a set of classical and predefined filters which are fast and easy to use.

First derivative filters

rcvSobel

Direct Sobel edges detection for image or matrix

function [src [image! vector!] dst [image! vector!] iSize [pair!] direction [integer!]]

src: image or matrix as vector

dst: image or matrix as vector

iSize: image or matrix size as pair

direction:

1: returns vertical gradient direction (Gx)

2: returns horizontal gradient direction (Gy)

3: both gradient directions by G= Gx + Gy

4: both gradients estimated by G= Sqrt (Gx^2 +Gy^2)

Used Hx, Hy and Ho (oblique) kernels

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & 2 \\ -1 & 0 & 1 \\ -2 & -1 & 0 \end{bmatrix}$$

```
img1: rcvLoadImage %../../images/lena.jpg
img2: rcvCreateImage img1/size
img3: rcvCreateImage img1/size
rcv2Gray/average img1 img2                  ; Grayscaled image
rcvSobel img2 img3 img1/size        4       ; Direct Sobel on image
```

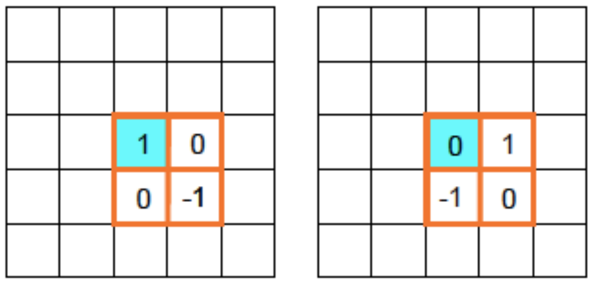
```
img1: rcvLoadImage %../../images/lena.jpg
img2: rcvCreateImage img1/size
mat1: rcvCreateMat 'integer! intSize img1/size
mat2: rcvCreateMat 'integer! intSize img1/size
rcvImage2Mat img1 mat1              ; Converts image to 1 Channel matrix [0..255]
rcvSobel mat1 mat2 img1/size        ; Sobel detector on Matrix
```

rcvRoberts

Robert's cross edges detection for image or matrix

rcvRoberts: function [src [image! vector!] dst [image! vector!] iSize [pair!] direction [integer!]
src: image or matrix as vector
dst: image or matrix as vector
iSize: image or matrix size as pair
factor: multiplier factor as float!
delta: bias for image brightness
direction:
1: returns vertical gradient direction (Gx)
2: returns horizontal gradient direction (Gy)
3: both gradient directions by G= Gx + Gy
4: both gradients estimated by G= Sqrt (Gx^2 +Gy^2)

Used Hx and Hy kernels



rcvPrewitt

Computes an approximation of the gradient magnitude of the input image

rcvPrewitt: function [src [image! vector!] dst [image! vector!] iSize [pair!] direction [integer!]
src: image or matrix
dst: image or matrix
iSize: size of the image or the matrix
direction:
1: returns vertical gradient direction (Gx)
2: returns horizontal gradient direction (Gy)
3: both gradient directions by G= Gx + Gy
4: both gradients estimated by G= Sqrt (Gx^2 +Gy^2)

Used Hx and Hy kernels

-1	0	1
-1	0	1
-1	0	1

-1	-1	-1
0	0	0
1	1	1

rcvKirsch

Computes an approximation of the gradient magnitude of the input image

rcvKirsch: function [src [image! vector!] dst [image! vector!] iSize [pair!] direction [integer!]
src: image or matrix
dst: image or matrix
iSize: size of the image or the matrix
direction:
1: returns vertical gradient direction (Gx)
2: returns horizontal gradient direction (Gy)
3: both gradient directions by G= Gx + Gy
4: both gradients estimated by G= Sqrt (Gx^2 +Gy^2)

Used Hx and Hy kernels

-3	-3	5
-3	0	5
-3	-3	5

-3	-3	-3
-3	0	-3
5	5	5

rcvGradNeumann

Computes the discrete gradient by forward finite differences and Neumann boundary conditions

rcvGradNeumann: function [src [image!] d1 [image!] d2 [image!]]
src: source image
d1: image for derivative along the x axis
d2: image derivative along the y axis

rcvDivNeumann

Computes the divergence by backward finite differences

rcvDivNeumann: function [src [image!] d1 [image!] d2 [image!]]
src: source image
d1: image for derivative along the x axis
d2: image derivative along the y axis

Second derivative filters

These filters use partial second derivative of an image or a matrix according to the equations

In x direction: $\left(\frac{\partial^2 I(x,y)}{\partial x^2}\right)$ and in Y direction: $\left(\frac{\partial^2 I(x,y)}{\partial y^2}\right)$

rcvDerivative2

A fast approximation of the second derivative of an image

rcvDerivative2: function [src [image! vector!] dst [image! vector!] iSize [pair!] factor [float!] direction [integer!]
src: image or matrix
dst: image or matrix
iSize: size of the image or the matrix
Factor: multiplier factor as float!
direction:
1: returns vertical gradient direction (Gx)
2: returns horizontal gradient direction (Gy)
3: both gradient directions by G= Gx + Gy

Used Hx and Hy kernels

	0	0	0	
	1	-2	1	
	0	0	0	

	0	1	0	
	0	-2	0	
	0	1	0	

rcvLaplacian

Computes the Laplacian of an image or matrix. The Laplacian is an approximation of the second derivative of an image

rcvLaplacian: function [src [image! vector!] dst [image! vector!] iSize [pair!] connexity [integer!]

src: image or matrix

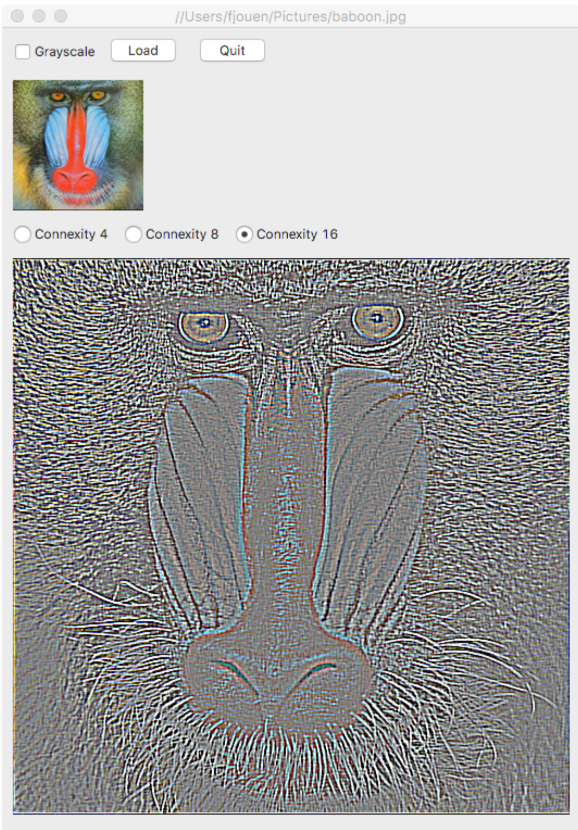
dst: image or matrix

iSize: size of the image or the matrix

connexity: neighbor pixels (4, 8 16)

Used kernels for 4, 8 or 16 connexity

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} \quad \begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix} \quad \begin{bmatrix} -1 & 0 & 0 & -1 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 \\ -1 & 0 & 0 & -1 \end{bmatrix}$$



Lines detection

RedCV includes Hough transform operator. The Hough transformation is a great way to detect lines in an image and it is quite useful for a number of computer vision tasks (see http://en.wikipedia.org/wiki/Hough_transform).
You can find here http://www.keymolen.com/2013/05/hough-transformation-c-implementation.html a very clear and detailed explanation. Thanks a lot to Bruno Keymolen for his original C++ code.

rcvMakeHoughAccumulator

Creates Hough accumulator
rcvMakeHoughAccumulator: func [w [integer!] h [integer!] return: [vector!]]
w: source image width
h: source image height

rcvGetAccumulatorSize

Gets Hough space accumulator size
rcvGetAccumulatorSize: function [acc [vector!] return: [pair!]

rcvHoughTransform

Makes Hough transform
rcvHoughTransform: function [mat [vector!] accu [vector!] w [integer!] h [integer!]]

rcvGetHoughLines

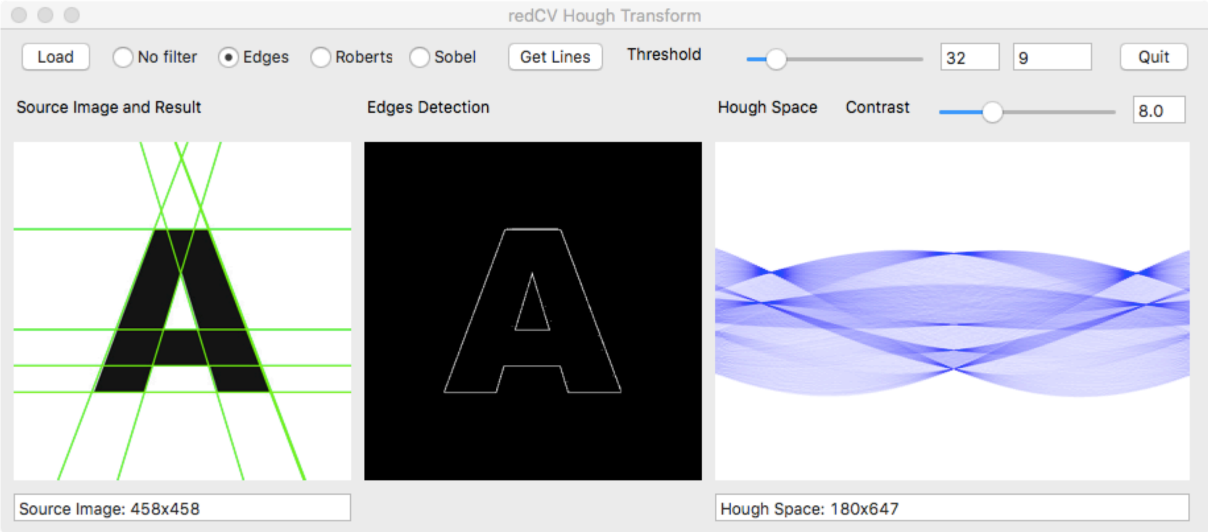
Gets lines in the accumulator according to threshold
rcvGetHoughLines: func [accu [vector!] img [image!] threshold [integer!] lines [block!]]

rcvHough2Image

Makes Hough space as red image
rcvHough2Image: function [mat [vector!] dst [image!] contrast [float!]]

Code Sample

```
rcv2BW edges bw                                  ; B&W image [0 255]
rcvImage2Mat bw mat                              ; B&W image to mat
acc: rcvMakeHoughAccumulator imgW imgH           ; makes Hough accumulator
rcvHoughTransform mat acc imgW imgH 128          ; performs Hough transform
hSpace: rcvCreateImage rcvGetAccumulatorSize acc ; creates Hough space image
rcvHough2Image acc hSpace contrast               ; shows Hough space
```



redCV Hough Transform
Load
No filter
Edges
Roberts
Sobel
Get Lines
Threshold
32
9
Quit
Source Image and Result
Edges Detection
Hough Space
Contrast
8.0
Source Image: 458x458
Hough Space: 180x647

Shape detection

RedCV includes Freeman chain code operator. The basic principle of chain codes is to separately encode each connected component, or "blob", in the image. For each such region, a point on the boundary is selected and its coordinates are transmitted. The encoder then moves along the boundary of the region and, at each step, transmits a number representing the direction of this movement.

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

You need a binary matrix [0..1] or [0..255] corresponding to your source image. See rcvMakeBinaryMat function.

rcvMatGetBorder

Gets pixels that belong to shape border

function [mat [vector!] matSize [pair!] value [integer!] border [block!]]
mat: source binary matrix
matSize: matrix size as pair
value: pixel value (default 1)
border: a block to store pixels direction

rcvMatGetChainCode

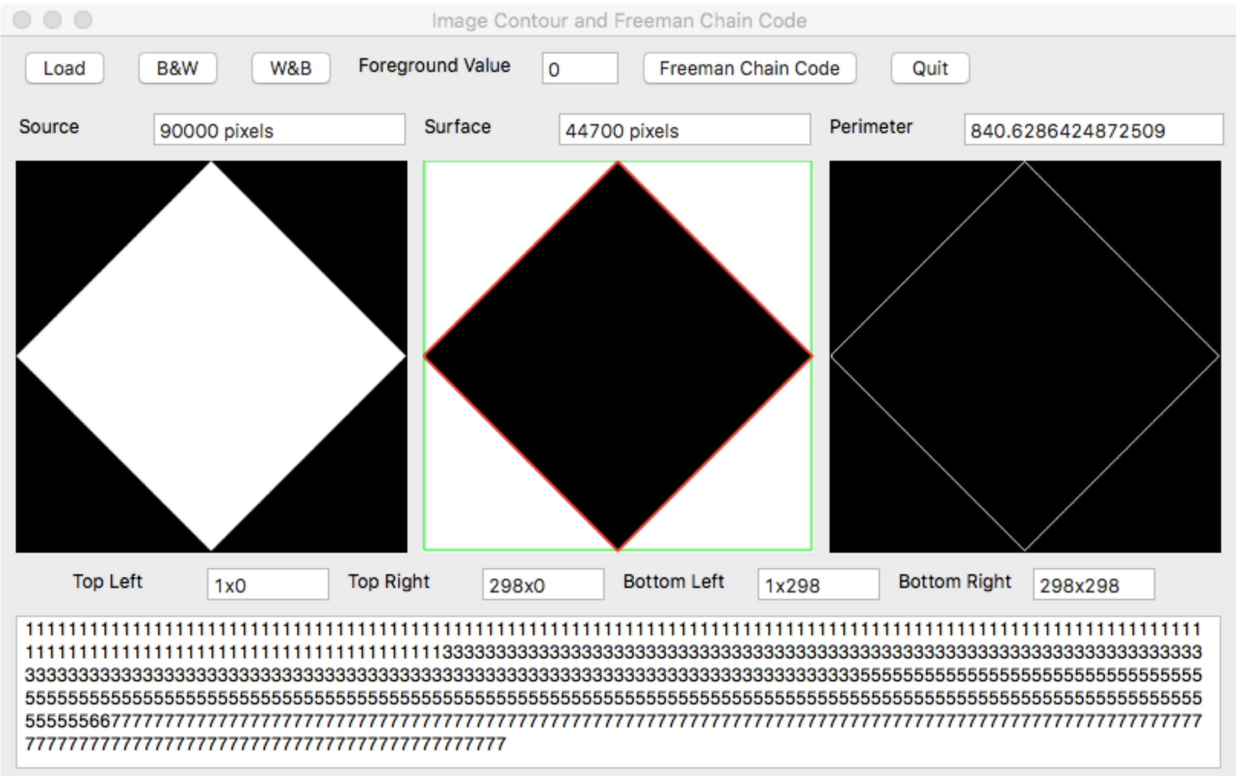
Gets Freeman Chain code

function [mat [vector!] matSize [pair!] coord [pair!] value [integer!] return: [integer!]
mat: source binary matrix
matSize: matrix size as pair
coord: pixel x and y coordinates
value: pixel value (default 1)

The function uses the current pixel and explores pixel neighbors in order to get the direction of the next pixel. Top-left pixel (first value of the border block) is used as starting coordinate, then the encoder clockwise processes next pixels.
ATTENTION: you have to write a function to store chain codes such as (see imagecontour.red sample. Visited is a temporary matrix used to store visited pixels)

```
getCodes: does [
	visited: rcvCreateMat 'integer! 32 matSize
	border: copy []
	rcvMatGetBorder bmat matSize 1 border
	foreach p border [rcvSetInt2D visited matSize p 255]
	count: length? border
	p: first border
	i: 1
```

```
		s: copy ""
		clear r/text
		perimeter: 0.0
		while [i < count] [
			d: rcvMatGetChainCode visited matSize p 255
			rcvSetInt2D visited matSize p 0 ; pixel is visited
			if d >= 0 [append s form d]; only external pixels -1: internal
			switch d [
				0	[p/x: p/x + 1 perimeter: perimeter + 1.0]		; east
				1	[p/x: p/x + 1 p/y: p/y + 1 perimeter: perimeter + sqrt 2]	; southeast
				2	[p/y: p/y + 1 perimeter: perimeter + 1.0]		; south
				3	[p/x: p/x - 1 p/y: p/y + 1 perimeter: perimeter + sqrt 2]	; southwest
				4	[p/x: p/x - 1 perimeter: perimeter + 1.0]		; west
				5	[p/x: p/x - 1 p/y: p/y - 1 perimeter: perimeter + sqrt 2]	; northwest
				6	[p/y: p/y - 1 perimeter: perimeter + 1.0]		; north
				7	[p/x: p/x + 1 p/y: p/y - 1 perimeter: perimeter + sqrt 2]	; northeast
			]
			i: i + 1
		]
]
```



Mathematical morphology

rcvCreateStructuringElement

The function allocates and fills a block, which can be used as a structuring element in the morphological operations

cvCreateStructuringElement: function [kSize [pair!] return: [block!] /rectangle /cross]
kSize: Kernel size (e.g. 3x3)

Refinement is used to create a cross-shaped element or a rectangular element

rcvErode

Erodes image by using structuring element

rcvErode: function [src [image!] dst [image!] kSize [pair!] kernel [block!]]
src: image
dst: image
kSize: kernel size as pair!
Kernel: block generated by cvCreateStructuringElement or customed structuring element

The function rcvErode erodes the source image using the specified structuring element that determines the shape of a pixel neighborhood over which the minimum is taken:

$$\texttt{dst=erode(src,element):}\quad \texttt{dst(x,y)=min}_{\texttt{((x',y') in element)}}\texttt{src(x+x',y+y')}$$

rcvErodeMat:

Erodes matrice by using structuring element

rcvErodeMat: function [src [vector!] dst [vector!] mSize [pair!] kSize [pair!] kernel [block!]]
src: matrix
dst: matrix
mSize: matrix size as pair!
kSize: kernel size as pair!

rcvDilate

Dilates image by using structuring element

rcvDilate: function [src [image!] dst [image!] kSize [pair!] kernel [block!]]
src: image
dst: image
kSize: kenel size as pair!
kernel: block generated by cvCreateStructuringElement or customed structuring element

The function rcvDilate dilates the source image using the specified structuring element that determines the shape of a pixel neighborhood over which the maximum is taken:

$$\texttt{dst=dilate(src,element):}\quad \texttt{dst(x,y)=max}_{\texttt{((x',y') in element)}}\texttt{src(x+x',y+y')}$$

rcvDilateMat

Dilates matrix by using structuring element

rcvDilateMat: function [src [vector!] dst [vector!] mSize [pair!] kSize [pair!] kernel [block!]]
src: matrix
dst: matrix
mSize: matrix size as pair!
kSize: kernel size as pair

rcvOpen

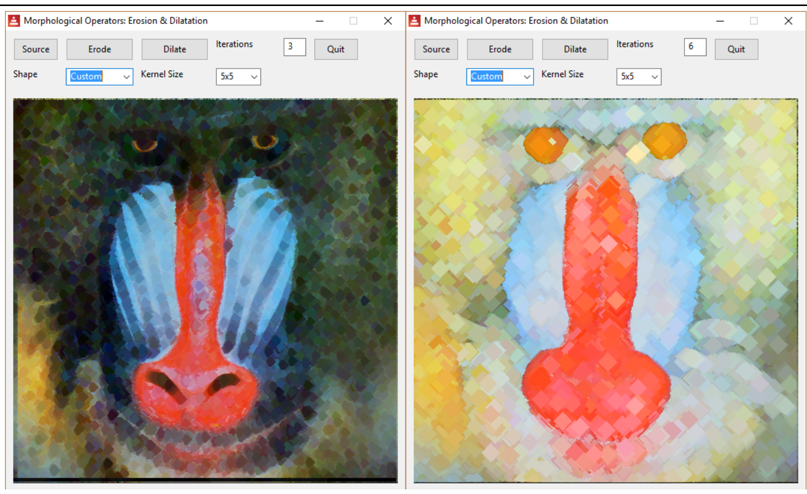
Erodes and Dilates image by using structuring element

rcvOpen: function [src [image!] dst [image!] kSize [pair!] kernel [block!]]
src: image
dst: image
kSize: kenel size as pair!
kernel: block generated by cvCreateStructuringElement or customed structuring element

rcvClose

Dilates and Erodes image by using structuring element

rcvClose: function [src [image!] dst [image!] kSize [pair!] kernel [block!]]
src: image
dst: image
kSize: kenel size as pair!
kernel: block generated by cvCreateStructuringElement or customed structuring element



Erosion　　　　Dilatation

rcvMGradient

Performs advanced morphological transformations using erosion and dilatation as basic operations

rcvMGradient: function [src [image!] dst [image!] kSize [pair!] kernel [block!] /reverse]
src: image
dst: image
kSize: kenel size as pair!
kernel: block generated by cvCreateStructuringElement or customed structuring element

dst=dilate src – erode src /reverse : dst=erode src – dilate src

rcvTopHat

Performs advanced morphological transformations

rcvTopHat: function [src [image!] dst [image!] kSize [pair!] kernel [block!]]
src: image
dst: image
kSize: kenel size as pair!
kernel: block generated by cvCreateStructuringElement or customed structuring element

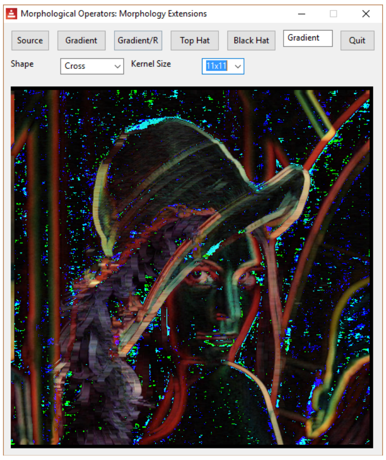
dst =src – open src dst

rcvBlackHat

Performs advanced morphological transformations

rcvTopHat: function [src [image!] dst [image!] kSize [pair!] kernel [block!]]
src: image
dst: image
kSize: kenel size as pair!
kernel: block generated by cvCreateStructuringElement or customed structuring element

dst = open src dst - src



rcvMMean

Means image by using structuring element

rcvMMean: function [src [image!] dst [image!] kSize [pair!] kernel [block!]]
src: image
dst: image
kSize: kernel size
kernel: block generated by cvCreateStructuringElement or customed structuring element

GUI Functions

Some functions for RedCV quick test. Functions are pure Red code. Routines are not required. These functions can also be used for displaying temporary images.

rcvNamedWindow

Creates a window

rcvNamedWindow: function [title [string!] return: [window!]]
title: Windows title as a string
window is returned a face datatype!

rcvDestroyWindow

Destroys a created window

rcvDestroyWindow: function [window [face!]]
window: Points to a window created by rcvNamedWindow

rcvDestroyAllWindows

Destroys all windows

rcvDestroyAllWindows: function []

rcvResizeWindow

Sets window size

rcvResizeWindow: function [window [face!] wSize [pair!]]

rcvMoveWindow

Sets window position

rcvMoveWindow: function [window [face!] position [pair!]]

rcvShowImage

Shows image in window

rcvShowImage: function [window [face!] image [image!]]

```
#include %../../libs/redcv.red
img1: rcvLoadImage %../../images/lena.jpg
s1: rcvNamedWindow "Source"
rcvShowImage s1 img1 wait 2
rcvMoveWindow s1 20x60 wait 2
rcvResizeWindow s1 512x512 wait 2
rcvDestroyWindow s1
do-events
```

Random Generator

Continuous Laws

randFloat

Returns a decimal value beween 0 and 1. Base 16 bit
randFloat: function[]

randUnif

Uniform law
randUnif: function [i [float!] j [float!]]
i: float value

randExp

Exponential law
randExp: function []

randExpm

Exponential law with a l degree
randExpm: function [l [float!]]
l: float value (e.g. 1.0)

randNorm

Normal law
randNorm : function [A [float!]]
A: float value (e.g. 1.0)

randLognorm

Lognormal law
randLognorm: function [a [float!] b [float!] z [float!]]
a: float value
b: float value
z: float value

randGamma

Gamma law
randGamma: func [k [integer!] l [float!] i]
k: integer value
l: float value

randDisc

Geometric law in a disc

randDisc: function []

randRect

Geometric law in a rectangle

randRect: function [a [float!] b [float!] c [float!] d [float!]]

a: float value

b: float value

c: float value

d: float value

randChi2

Chi square law

randChi2: function [v [integer!]]

v: integer value (e.g. 2)

randErlang

Erlang law

randErlang: function [n [integer!]]

n: integer value (e.g. 2)

randStudent

Student law

randStudent: function [n [integer!] z [float!]]

n: integer value (e.g. 3)

z: float value (e.g. 1.0)

randFischer

Fisher law (e.g 1 1)

randFischer: function [n [integer!] m [integer!]]

n: integer value (e.g. 1)

m: integer value (e.g. 1)

randLaplace

Laplace law

randLaplace: function [a [float!] /local u1 u2]

a: float value (e.g. 1.0)

randBeta

Beta law

randBeta: function [a [integer!] b [integer!]]
a: integer value (e.g. 1)
b: integer value (e.g. 1)

randWeibull

Weibull law

randWeibull: function [a [float!] l [float!]]
a: float value (e.g. 1.0)
l: float value (e.g. 1.0)

randRayleigh

Rayleigh law

randRayleigh: function []

Discrete Laws

randBernouilli

Bernouilli law

randBernouilli: function [p [float!]]
p: float value (e.g. 0.5)

randBinomial

Binomial law

randBinomial: function [n [integer!] p [float!]]
n: integer value (e.g. 1)
p: float value (e.g. 0.5)

randBinomialneg

Binomial negative law (e.g. 1 0.5)

randBinomialneg: function [n [integer!] p [float!]]
n: integer value (e.g. 1)
p: float value (e.g. 0.5)

randGeo

Geometric law

randGeo: func [p [float!]]
p: float value (e.g. 0.25)

randPoisson

Poisson law

randPoisson: function [l [float!]]

l: float value (e.g. 1.5)