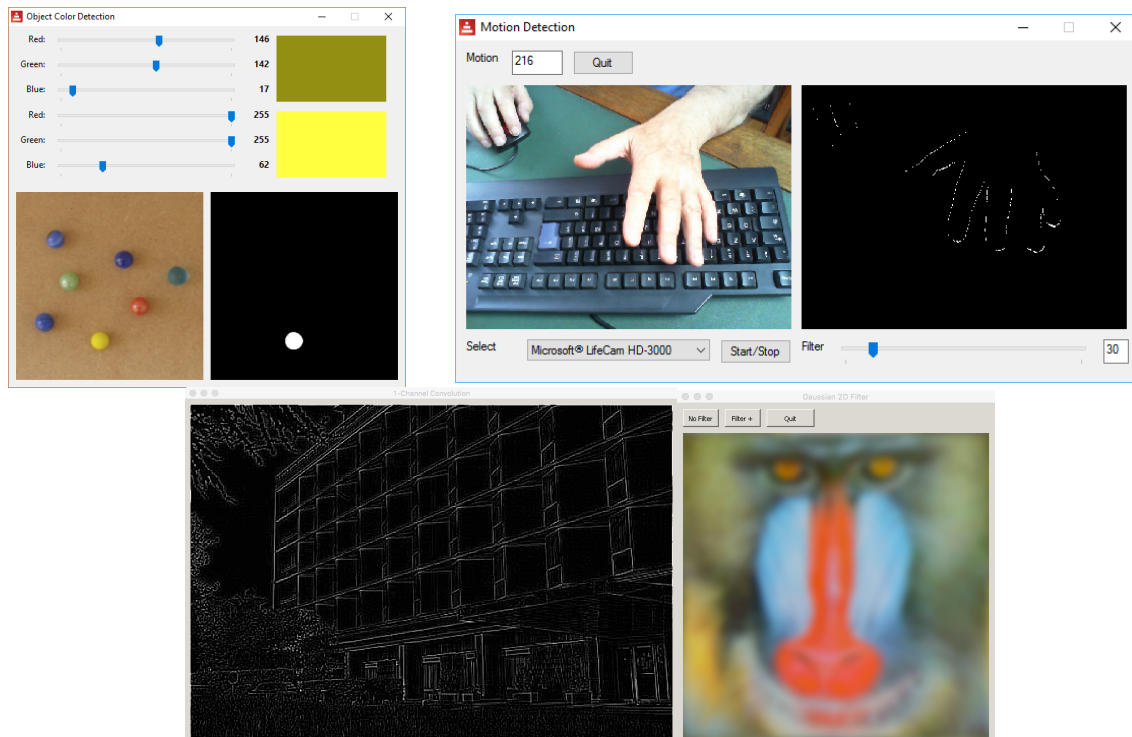


# RedCV Open Source Computer Vision Library



## 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 about 145 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.

Tarek Hamouda for the library optimization.

Thanks to Nénad Rakocevic and Qingtian Xie for developing Red and their constant help.

Thanks to Didier Cadieu for samples optimization.

## Where to get RedCV

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

## 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
]
```

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

# *RedCV Reference Manual*

- Using RedCV library
- Basic structures
- Operations on arrays
- Logical operators on arrays
- Mathematical operators on arrays
- Statistical operators on arrays
- Random generator
- Arrays conversions
- Filters
- Color conversions
- Operations on sub-arrays
- Image transformations
- Highgui functions

# 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.

Channels can be easily accessed by a pointer

```
pixel >>> 24           : Alpha (transparency) channel
pixel and FF0000h >> 16 : Red channel
pixel and FF00h >> 8    : Green channel
pixel and FFh           : Blue channel
```

## Matrix

Matrix! Datatype is not yet implemented by Red. We simulate matrices with Red vector! datatype. Work under progress. Matrices are 2-D with n lines \*m columns. Matrix element can be Char!, Integer! or Float!

# Operations on Arrays

## Images

### 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
```

### rcvReleaseImage

**Releases image data**

rcvReleaseImage: function [src [image!]]

src : image to remove

This function will be probably removed with 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]
```

### rcvLoadImageB

**Load image from file and return image as binary**

rcvLoadImageB: 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

### rcvCloneImage

**Returns a copy of source image**

rcvCloneImage: function [src [image!]] return: [image!]]

src: image to be cloned

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

## 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
```

## rcvZeroImage

### 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

## 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]

## Matrices

## 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
```

## rcvReleaseMat

### Releases Matrix

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

## rcvCloneMat

### Returns a copy of source matrix

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

## rcvCopyMat

### Copy source matrix to destination matrix

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

## rcvRandomMat

### Randomize matrix

rcvRandomMat: function [mat [vector!] value [integer!]]  
mat: destination 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 FFFFFFFh
```

## 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 ;
```

# Logical Operators on Arrays

## Images

### rcvAND

**dst: src1 AND src2**

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

src1: first image

src2: second image

dst: src1 and src2

### rcvOR

**dst: src1 OR src2**

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

src1: first image

src2: second image

dst: src1 or src2

### rcvXOR

**dst: src1 XOR src2**

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

src1: first image

src2: second image

dst: src1 xor src2

### 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**

rcvNXOR: 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!

# Matrices

## rcvANDMat

**Returns source1 AND source2**

rcvAndMat: function [src1 [vector!] src2 [vector!] return: [vector!]

src1: first matrice

src2: second matrice

## rcvORMat

**Returns source1 OR source2**

rcvORMat: function [src1 [vector!] src2 [vector!] return: [vector!]

src1: first matrice

src2: second matrice

## rcvXORMat

**Returns source1 XOR source2**

rcvORMat: function [src1 [vector!] src2 [vector!] return: [vector!]

src1: first matrice

src2: second matrice

## 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!

# Mathematical Operators on Arrays

## Images

### rcvAdd

**dst: src1 + src2**

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

src1: image

src2: image

dst: image

### rcvSub

**dst: src1 - src2**

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

src1: image

src2: image

dst: image

### rcvMul

**dst: src1 \* src2**

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

src1: image

src2: image

dst: image

### rcvDiv

**dst: src1 / src2**

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

src1: image

src2: image

dst: image

### 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

## 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: maximum src1 src2**

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

src1: image

src2: image

dst: image

## rcvAddS

**dst: src + integer! value**

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

src: image

dst: image

val: integer

## rcvSubS

**dst: src - integer! value**

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

src: image

dst: image

val: integer

## rcvMulS

**dst: src \* integer! value**

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

src: image

dst: image

val: integer

## rcvDivS

**dst: src / integer! value**

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

## rcvModS

**dst: src // integer! Value (modulo)**

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

## rcvRemS

**dst: src % integer! Value (remainder)**

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

## 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**

```
rcvRSH: function [src [image!] dst [image!] val [integer!]]  
src: image  
dst: image  
val: integer
```

## rcvPow

**dst: src ^integer! Value**

```
rcvPow: function [src [image!] dst [image!] val [integer!]]  
src: image  
dst: image  
val: integer
```

## rcvSqr

**Image square root**

```
rcvSqr: function [src [image!] dst [image!] val [integer!]]
```

src: image  
dst: image  
val: integer

### rcvAddT

**dst: src + tuple! value**

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

### rcvSubT

**dst: src - tuple! value**

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

### rcvMulT

**dst: src \* tuple! value**

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

### rcvDivT

**dst: src / tuple! value**

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

### rcvModT

**dst: src // tuple! Value (modulo)**

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

### rcvRemT

**dst: src % tuple! Value (remainder)**

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

## Matrices

### rcvAddMat

**dst: src1 + src2**

rcvAddMat: function [src1 [vector!] src2 [vector!] return: [vector!]]

src1: first matrix

src2: second matrix

### rcvSubMat

**dst: src1 - src2**

rcvSubMat: function [src1 [vector!] src2 [vector!] return: [vector!]]

src1: first matrix

src2: second matrix

## rcvMulMat

**dst: src1 \* src2**

rcvMulMat: function [src1 [vector!] src2 [vector!] return: [vector!]]

src1: first matrix

src2: second matrix

## rcvDivMat

**dst: src1 / src2**

rcvDivMat: function [src1 [vector!] src2 [vector!] return: [vector!]]

src1: first matrix

src2: second matrix

## rcvRemMat

**dst: src1 % src2**

rcvRemMat: function [src1 [vector!] src2 [vector!] return: [vector!]]

src1: first matrix

src2: second matrix

## rcvAddSMat

**src + value**

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

src: matrix

value: integer

## rcvSubSMat

**src - value**

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

src: matrix

value: integer

## rcvDivSMat

**src / value**

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

src: matrix

value: integer

## rcvRemSMat

**src % value (remainder)**

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

src: matrix

value: integer

# Statistical Operators on Arrays

## Images and Matrices

### 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

### 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

### 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

### 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

## 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

## rcvHistogram

Calculates array histogram

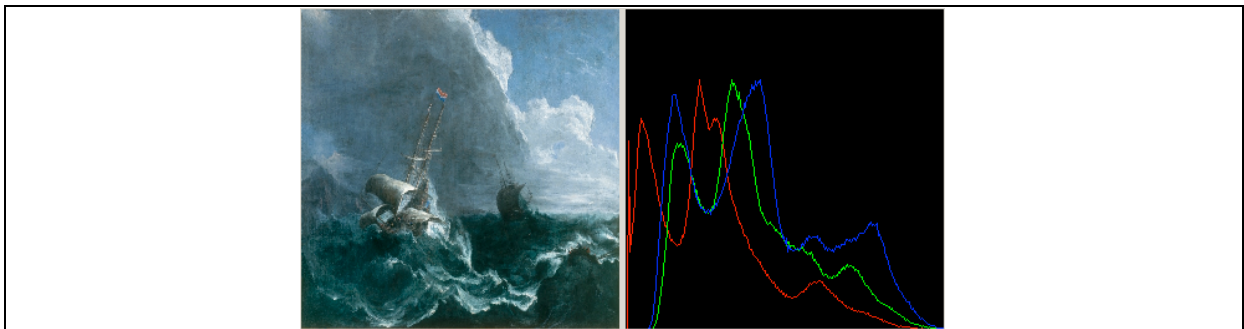
rcvHistogram: function [arr [image! vector!] return: [vector!] /red /green /blue]

arr: image or vector

/red: histogram for red channel

/green: histogram for green channel

/blue: histogram for blue channel



## rcvRangeImage

Gives range value in Image as a tuple

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

source: image

## rcvSortImage

Ascending image sorting

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

source: image

dst: image

# Random Generator

## Continuous Laws

### randFloat

Returns a decimal value between 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

## randBernoulli

### Bernoulli law

randBernoulli: 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)

# Arrays Conversions

## Images

### rcv2BW

**Convert RGB image to Black[0] and White [255]**

rcv2BW: function [src [image!] dst [image!]]

src: source image

dst: destination image

### rcv2Gray

**Convert 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

### rcvInvert

**Destination image: inverted source image (Similar to NOT image)**

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

src: source image

dst: destination image

## rcvImage2Mat

### Converts Red Image to 8-bit 2-D Matrix

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

## Matrices

## rcvMat82Image

### 8-bit Matrix to Red Image

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

## rcvMat162Image

### 16-bit Matrix to Red Image

rcvMat162Image: function [mat [vector!] dst [image!]]  
mat: vector  
dst: image

## rcvMat322Image

### 32-bit Matrix to Red Image

rcvMat322Image: 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 FFFFFFFh
```

## 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!

# Filters

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.

## General functions

### 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  $\sigma$  is the standard deviation of the distribution.

### rcvConvolve

**Convolve 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

## 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

## Images and matrices Filters

### rcv2BWFilter

**Binarization of RGB image image according to threshold**

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

src: image

dst: image

thresh: threshold integer value

### 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

## 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 1 1x1 1 rcvFilter2D src dst

## rcvSobel

### Direct Sobel edges detection for image or matrix

```
rcvSobel: function [src [image! vector!] dst [image! vector!] iSize [pair!]]
src: image or matrix as vector
dst: image or matrix as vector
iSize: image or matrix size as pair
```

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

## rcvRobert

### Robert's cross edges detection for image or matrix

```
rcvRobert: function [src [image! vector!] dst [image! vector!] iSize [pair!] factor [float!] delta
[!float!]]
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

# Color Conversions

## Images

### 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 360°. 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

### RBG color to YCrCb conversion

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

There is no transformation matrix.

$$Y <- 0.299 * R + 0.587 * G + 0.114 * B$$
$$Cr <- (R - Y) * 0.713 + \text{delta}$$
$$Cb <- (B - Y) * 0.564 + \text{delta}$$

## 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
```

## rcvBGR2Lab

### 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  $0 \leq L \leq 100$ ,  $-127 \leq a \leq 127$ ,  $-127 \leq b \leq 127$ . The values are then converted to 8-bit images:  $L \leftarrow L * 255 / 100$ ,  $a \leftarrow a + 128$ ,  $b \leftarrow b + 128$ .

## rcvRGB2Luv

### RGB color to CIE L\*u\*v conversion

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

src: image

dst: image

## rcvBGR2Luv

### RGB 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  $0 \leq L \leq 100$ ,  $-134 \leq u \leq 220$ ,  $-140 \leq v \leq 122$ . The values are then converted to 8-bit images:  $L \leftarrow L * 255 / 100$ ,  $u \leftarrow (u + 134) * 255 / 354$ ,  $v \leftarrow (v + 140) * 255 / 256$ .

# Operations on Sub-Arrays

## Images and Matrices

### rcvSplit

**Separates source image in RGBA 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

### rcvInRange

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

rcvInRange: function [src [image!] dst [image!] lower [tuple!] upper [tuple!]]

src: source image

dst: destination image

lower: lower tuple

upper: lower tuple





# Image Transformations

## rcvFlip

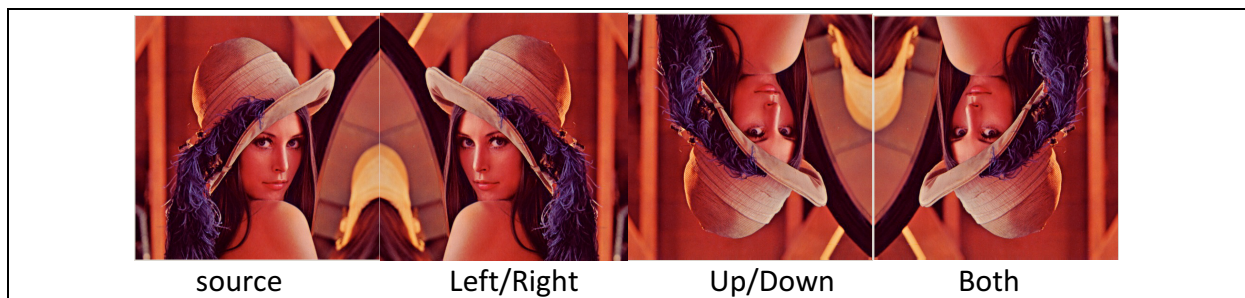
**Left/Right, Up/Down or both directions image flip**

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!] canvas 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 canvas nSize
```

# Highgui Functions

Some highgui 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
```