IMLAB Image Processing and Matrix Library

Generated by Doxygen 1.8.5

Mon Dec 3 2018 19:32:43

Contents

Chapter 1

IMLAB Image Processing Library

IMLAB is an easy to use computer vision library written purely in C language. It is designed to be tiny, simple and brief yet self contained. No addtitional libraries is needed. Most of the fundemental operations on images, vectors, matrice and files are written pureley in C as header files which can be used sepeartely depending on your project specifics. Basically, IMLAB defines just a few very well known structures in order to keep the library readable and work comfortable. It has 5 main core project all under the imlab and all of them can be included or excluded from the main projet depending on your task. Although the library is written in C, it is designed to be look as object oriented manner so it is easy to read and it is easy to remember neccesarry functions for each operation. Each struct (will menteioned as class later) has its own operations and all operation belong to a Class is started with a class_name prefix. This type of access makes it possiple to use auto complete features in the editors and helps to find the realted functions easily.

IMLAB defines the following structure:

- vector_t is a vector container just like standart std vector in C++. vector_t structure is a generic c-like array that can hold any type of variable in dynamically allocated memory. Just like the arrays, vector data is contigous in memory and can be accessed externally. Using a vector object is easy. Just decide the data type you need and call vector_create to create a new vector object. The data type can be any C data types, additional names declared in stdint.h or structs created by the user.

  ```
  vector_t *out = vector_create(uint32_t);
  ```

  The above code will create a vector object and set the initial length to zero and capacity to one. In order to add/remove element to vector vector_push/vector_pop can be used.

  ```
  vector_t out = vector_create(float);
  float values[12] = {0.0 0.1, 0.3, 0.5, 0.7, 0.9, 0.8, 0.6, 0.5, 0.4, 0.2, 0.0};
  // in this loop the length of the vector will automatically increase up to 12.
  for(size_t i = 0; i < 12; i++) {
   vector_push(out, &values[i]);
  }
  ```

  We can get the values from the vector with various methods. In the simplest case if we know the data type of the vector, we can obtain the data pointer and access the values via pointer just like standart c arrays.

  ```
  float *vector_data = data(float, out);
  for(size_t i = 0; i < length(out); i++) {
     float val = vector_data[i];
  }
  ```

- matrix_t structure is a generic c-like array that can hold any type of variable in dynamically allocated memory. Just like the arrays and vectors, matrix data is contigous in memory and can be accessed externally. Similar to vector objects, matrix object is also resizable but you cannot increase the size of the matrix by pushing elements. The main advantage of the matrix object onto the standart arrays is that the matrix objects can hold two dimesional data in a simple way. Using matrix object the user inetarctions with the 2D array becomes easier and the code would be understandable.

  Similar to the vector_t structure matrix_t structure can be created via matrix_create with any C data types, additional names declared in stdint.h or structs created by the user.

```
matrix_t *out = matrix_create(float);
```

The above code will create a matrix object and set the initial rows,cols and channels to zero. If you want to specify size to matrix just call:

```
matrix_t *out = matrix_create(float, 1024, 1024, 1);
```

This call of the matrix_create will create a floating point data pointer and allocate 1024*1024*1*sizeof(float) byte memory on the memory. In order to access an element of a matrix at or data functions could be used.

```
for(size_t i = 0; i < rows(out); i++) {
   for(size_t j = 0; j < cols(out); j++) {
       float val = at(float, out, i,j);
   }
}
```

Links are generated automatically for webpages (like `http://www.google.co.uk`) and for structures, like BoxStruct_struct. For typedef-ed types use #BoxStruct. For functions, automatic links are generated when the parenthesis () follow the name of the function, like Box_The_Function_Name(). Alternatively, you can use #Box_-The_Function_Name.

Returns

`NULL` is always returned.

Chapter 2

Data Structure Index

2.1 Data Structures

Here are the data structures with brief descriptions:

Chapter 3

File Index

3.1 File List

Here is a list of all files with brief descriptions:

Chapter 4

Data Structure Documentation

4.1 color_t Struct Reference

```
#include <imcore.h>
```

Data Fields

- uint8_t blue
- uint8_t green
- uint8_t red

4.1.1 Field Documentation

4.1.1.1 uint8_t color_t::blue

4.1.1.2 uint8_t color_t::green

4.1.1.3 uint8_t color_t::red

The documentation for this struct was generated from the following file:

- E:/imlab_library/include/imcore.h

4.2 disjoint_set_t Struct Reference

```
#include <alcore.h>
```

Data Fields

- uint32_t length
- uint32_t * parent
- uint32_t * label
- int32_t * rank

4.2.1 Field Documentation

4.2.1.1 uint32_t* disjoint_set_t::label

4.2.1.2 uint32_t disjoint_set_t::length

4.2.1.3 uint32_t* disjoint_set_t::parent

4.2.1.4 int32_t* disjoint_set_t::rank

The documentation for this struct was generated from the following file:

- E:/imlab_library/include/alcore.h

4.3 feature_t Struct Reference

```
#include <cvcore.h>
```

Data Fields

- enum cv_algorithm_t algorithm
- uint32_t feature_size
- uint32_t image_width
- uint32_t image_height
- void * parameters
- return_t(* method)(matrix_t *, struct feature_t *, float *)

4.3.1 Field Documentation

4.3.1.1 enum cv_algorithm_t feature_t::algorithm

4.3.1.2 uint32_t feature_t::feature_size

4.3.1.3 uint32_t feature_t::image_height

4.3.1.4 uint32_t feature_t::image_width

4.3.1.5 return_t(* feature_t::method)(matrix_t *, struct feature_t *, float *)

4.3.1.6 void* feature_t::parameters

The documentation for this struct was generated from the following file:

- E:/imlab_library/include/cvcore.h

4.4 glm_t Struct Reference

glm_t struct keeps the given GLM parameters in a single variable and computes the necessary parameters at the construction step.

```
#include <mlcore.h>
```

4.4.1 Detailed Description

glm_t struct keeps the given GLM parameters in a single variable and computes the necessary parameters at the construction step.

Generalized Linear Models are machine learning techniques that approximates the output labels y as the following linear form.

$$y = w \times x + b$$

Here w is the linear coefficient and b is the bias. The problem of the GLM are to find the best coefficient vector and bias value depending on the given restrictions on these vectors. In the most generalized way the cost function can be defined as the sum of the classification and regularization loss

$$C(w,b) := \frac{1}{n}\sum_i L(y_i, a_i = w_i \times x_i + b) + R(w, \lambda)$$

Using the above cost function one can create different sets of problems/solutions by simply changing the lost function L and regularization function R. Here a list of supported lost and regularization functions are given.

algorithm	**lost function** (L)	**regularization** (R)
Least-squares without regularization	$L = (a-y)^2$	$R = 0$
Least-squares using a ridge (L1) penalty	$L = (a-y)^2$	$R = \lambda\|w\|$
Least-squares using a lasso (L2) penalty	$L = (a-y)^2$	$R = \lambda w^2/2$
Logistic regression without regularization	$L = \log(1+\exp(-ay))$	$R = 0$
Logistic regression using a L1 penalty	$L = \log(1+\exp(-ay))$	$R = \lambda\|w\|$
Logistic regression using a L2 penalty	$L = \log(1+\exp(-ay))$	$R = \lambda w^2/2$
Support Vector Machine without regularization	$L = \max(0, 1-ay)$	$R = 0$
Support Vector Machine using a L1 penalty	$L = \max(0, 1-ay)$	$R = \lambda\|w\|$
Support Vector Machine using a L2 penalty	$L = \max(0, 1-ay)$	$R = \lambda w^2/2$

Supported Algorithms

Logistic Regression

Logistic regression is a special case of generalized linear models. These models assume that the data points are generated from a non-linear transformation of a linear function. If we call this non-linear function as $y = \theta(s)$, s will be the output of the linear function

$$s = w^T x$$

where $x_n \in R^d$ correspond to input vector with d dimensions and $y_n \in \{-1, +1\}$ is the associated label for the data vector x_n. Here if the $\theta(s)$ is selected as the

$$\theta(s) = s$$

the problem turns into the liner regression where the output has no bounds and for the

$$\theta(s) = sign(s)$$

the model becomes the linear classifier where the outputs are -1 and 1.

For the logistic regression, the non-linear function is selected as

$$\theta(s) = \frac{e^s}{1+e^s}$$

which we can pretend to outputs as the probability values. Here we can express the above equation in terms of probabilities as

$$P(y=1|x)=\theta(s)=\frac{1}{1+e^{-w^Tx}}$$

and

$$P(y=-1|x)=1-\theta(s)=\frac{e^{-w^Tx}}{1+e^{-w^Tx}}=\frac{1}{1+e^{w^Tx}}$$

Note that we derive the second equation using the fact that the sum of two probabilities $P(y=1|x)$ and $P(y=0|x)$ must be one. We can further simplify the likelihood using the fact that $\theta(s)=1-\theta(-s)$, so the probabilities becomes

$$P(y|x)=\frac{1}{1+e^{-yw^Tx}}$$

Here we try to find such weights w which maximizes these probabilities for all samples $n=1,2,\ldots,N$. The problem can be expressed in terms of likelihood function as

$$L(x,y|w)=\Pi_{n=1}^{N}P(y|x)=\Pi_{n=1}^{N}\frac{1}{1+e^{-yw^Tx}}$$

where we try to find w which maximizes $L(x,y|w)$. Since the likelihood function involves exponentials, using log-likelihood of the data create more easy-to-solve equation. In this case the log-likelihood function becomes

$$L(x,y|w)=\sum_{n=1}^{N}\log\left(\frac{1}{1+e^{-yw^Tx}}\right)$$

and maximization of this log-likelihood function is equivalent to minimize

$$E(x,y|w)=\sum_{n=1}^{N}\log\left(1+e^{-yw^Tx}\right)$$

Note that the resulting error function has not a closed-form which the solution need be found using an iterative algorithm. Moreover, the penalty for the large weights must also be considered to avoid over fitting to data. In this thesis, following **[towards]** to automated caricature}, L_2 regularized logistic regression is used **[Dual]** Coordinate Descent Methods for Logistic Regression}, so the objective function become

$$\arg\min_{w}\sum_{n=1}^{N}\log\left(1+e^{-yw^Tx}\right)+\frac{\lambda}{2}\sum_{n=1}^{N}w_n^2$$

where the λ is the trade off between matching the training data and the generalization. To solve the above equation we use the gradient descent method

$$w_k^{(t+1)}=w_k^{(t)}-\varepsilon\frac{\partial E}{\partial w}$$

where the gradient is

$$\frac{\partial E}{\partial w}=-\sum_{n=1}^{N}yx\log\left(1+e^{-yw^Tx}\right)+\lambda\sum_{n=1}^{N}w_n$$

Support Vector Machine (SVM)

Support Vector Machine (SVM) is a successful learning algorithm that has many uses in pattern recognition for classification and regression **[cortes1995support]**,moghaddam2000gender}. The basic idea behind the SVM is finding a hyper-plane or hyper-planes which separate the two classes with a maximum margin, transforming the problem into a quadratic optimization problem. For the problems that cannot be linearly separated in the input space, SVM maps the input space into a much higher-dimensional feature space, where separation is possible. The power of SVM stems from the principle of structural risk minimization that while increasing the classification success, keeps down the VC dimension as much as possible, so that the generalization of the data is maintained. In other words, classification by mapping the data into high-dimensional space and maximizing margin there is the proof of the founded hyper-plane is the best descriptor for that data rather than over-fitting (see Fig. fig:svm}).

The basic mechanism of SVM may be stated as follows: Given a labeled set of N samples (x_n, y_n) where $x_n \in R^d$ correspond to input vectors and $y_n \in \{-1, +1\}$ associated label for current vector **[parsons2005introduction]**}. A linear classification problem may be expressed in terms of our notation as follows:

$$f(\mathbf{x}) = \mathbf{w}^T \phi(\mathbf{x}) + b$$

where $\phi(\mathbf{x})$ denotes to the feature-space transformation function that makes the data vector $\mathbf{x}$ linearly separable in transform domain, $\mathbf{w}$ is the weights for the data in feature space and b is the bias term. Assuming that the data is linearly separable in feature space, there is at least one $\mathbf{w}$ and b that satisfies $f(\mathbf{x_n}) > 0$ for $y_n = +1$ and $f(x_n) < 0$ for $y_n = -1$, note that $y_n f(x_n) > 0$ for all training points.

There might be of course many hyper-plane created by different $\mathbf{w}$ and b, SVM aims to find that gives the smallest generalization error by making the hyper-plane as far as possible to the nearest data points also known as support vectors **[parsons2005introduction]**}. The distance of a point x_n to the hyper-plane which is defined by $\mathbf{w}^T \phi(\mathbf{x}) + b = 0$ is:

$$\frac{|f(x_n)|}{\|\mathbf{w}\|} = \frac{y_n(\mathbf{w}^T \phi(x_n) + b)}{\|\mathbf{w}\|}$$

Note that $|f(\mathbf{x})|$ can be replaced by $y_n f(\mathbf{x})$ under the condition that all the data points are correctly classified.

In order to maximize the distance between hyper-plane and the nearest data point, assuming that the x_n is the closest point to the plane,we are looking for the arguments

$$\arg\min_{\mathbf{w},b} \left\{ \frac{1}{\|\mathbf{w}\|} \min_n \left[y_n(\mathbf{w}^T \phi(x_n) + b) \right] \right\}$$

Because of the complexity of direct solution of the problem, the equation may be expressed as an equivalent problem. Multiplying the distances by a constant in scale domain does not change the problem, so using this as an advantage, we can find a scaling constant that assures:

$$y_n(\mathbf{w}^T \phi(x_n) + b) = 1$$

Note that this scaling also assures that:

$$y_n(\mathbf{w}^T \phi(x_n) + b) \geq 1$$

In order to solve maximization problem, the problem is quickly transformed into quadratic minimization problem:

$$\arg\min_{\mathbf{w},b} \frac{1}{2} \mathbf{w}^T \mathbf{w}$$

Note that we still solve the equivalent problem under the constrain that $y_n(\mathbf{w}^T \phi(x_n) + b) \geq 1$. One can simply notice that this constrained optimization problem may easily be solved using Lagrange multipliers

$$L(\mathbf{w}, b, \mathbf{a}) = \frac{1}{\|\mathbf{w}\|^2} - \sum_{n=1}^{N} a_n \left\{ y_n(\mathbf{w}^T \phi(x_n) + b) - 1 \right\}$$

Here a_n is an non-negative multiplier for each constrain, for the minus sign in front of the a_n, problem can also be considered as an maximization problem with respect to the a_n. In order to find a solution to this quadratic problem, derivation of the equation with respect to $\mathbf{w}$ and b must be set to zero.

$$\frac{\partial L}{\partial \mathbf{w}} = \mathbf{w} - \sum_{n=1}^{N} a_n y_n \phi(x_n)$$

$$\frac{\partial L}{\partial b} = -\sum_{n=1}^{N} a_n y_n$$

Substituting these equations into the original one, we come up with a dual representation problem:

$$\hat{L}(a) = \sum_{n=1}^{N} a_n - \frac{1}{2}\sum_{n=1}^{N}\sum_{m=1}^{N} a_n a_m y_n y_m \phi(x_n)^T \phi(x_m)$$

which maximize the margin with respect to the a_n under the following constrains:

$$a_n \geq 0, \quad for\, n = 1, 2, ...N$$

$$\sum_{n=1}^{N} a_n y_n = 0$$

Note that the $\phi(x)$ could be a transform function that represent inputs in an infinite dimension space. However the solution of the problem only needs the dot product of $\phi(x_n)^T \phi(x_m)$, so for the solution of the problem, the direct computation of the high dimension feature space is not necessary. Defining a $K(x_n, x_m)$ kernel function which consist of dot products of the input vector, the problem could be solved in low complexity. Definition of the $\phi(x)$ function determines the kernel function, some of the kernel function frequently used with SVM given as follows:

kernel	formula
Linear	$K(x_n, x_m) = x_n^T x_m$
Polynomial	$K(x_n, x_m) = (x_n^T x_m + c)^d$
Radial Basis	$K(x_n, x_m) = \exp(\frac{\|x_n - x_m\|_2^2}{\sigma^2}$

The $a_n \geq 0$'s coming from the solution shows the support vectors, so the equation of the hyper-plane is defined by:

$$\mathbf{w} = \sum_{n \in SV} a_n y_n x_n$$

The documentation for this struct was generated from the following file:

- E:/imlab_library/include/mlcore.h

4.5 haar_t Struct Reference

```
#include <cvcore.h>
```

Data Fields

- int size1
- int size2
- int length
- struct haar_stage_t ** stages

4.5.1 Detailed Description

Haar cascade holder

4.5.2 Field Documentation

4.5.2.1 int haar_t::length

4.5.2.2 int haar_t::size1

4.5.2.3 int haar_t::size2

4.5.2.4 struct haar_stage_t** haar_t::stages

The documentation for this struct was generated from the following file:

- E:/imlab_library/include/cvcore.h

4.6 matrix_t Struct Reference

matrix_t is a n-channel 2D matrix container.

`#include <core.h>`

Data Fields

- struct imlab_type_t * _type
- uint32_t _rows
- uint32_t _cols
- uint32_t _channels
- void * _data

4.6.1 Detailed Description

matrix_t is a n-channel 2D matrix container.

matrix_t structure is a generic c-like array that can hold any type of variable in dynamically allocated memory. Just like the arrays and vectors, matrix data is contigous in memory and can be accessed externally. Similar to vector objects, matrix object is also resizable but you cannot increase the size of the matrix by pushing elements. The main advantage of the matrix object onto the standart arrays is that the matrix objects can hold two dimensional data in a simple way. Using matrix object the user interactions with the 2D array becomes easier and the code would be understandable.

IMLAB defines the following functions for vector objects:

- width : returns the number of coloumns of the matrix
- cols : returns the number of coloumns of the matrix
- height : returns the number of rows of the matrix
- rows : returns the number of rows of the matrix
- channels : returns the number of channels of the matrix
- volume : returns the volume (rows*cols*channels) of the matrix
- mdata / data : returns the pointer to the data of the matrix
- matrix_create : constructor function zero initializer
- matrix_resize : change the size of the data holded in container
- matrix_free : destructor functions
- matrix_fill : set the struct with a constant value
- matrix_read : loads the matrix struct from the hard drive (uncompress data)
- matrix_write : saves the type struct to the hard drive (compress data)

- matrix_copy : creates a copy of the input matrix (size and data) and returns it

Using a vector object is easy. Just decide the data type you need and call vector_create to create a new vector object. The data type can be any C data types, additional names declared in stdint.h or structs created by the user.

```
matrix_t *out = matrix_t(uint32_t);
```

The above code will create a matrix object and set the initial rows, cols and channels to zero and data pointer to the NULL. If you want to specify an initial size for the matrix just call:

```
matrix_t out = matrix_create(float, 1024, 1024);
```

This call of the matrix_create will create a floating point data pointer and allocate 1024∗1024∗sizeof(float) byte memory on the memory. If you want to marix to have more than one channels, you can add the third term as the number of channels. It is also possible to create a matrix from an existing ones or from c pointers. In this case the fourth argument should be the pointer to the data:

```
double numbers[9] = {1,2,3, 4,5,6, 7,8,9};
matrix_t *out = matrix_create(double, 3,3,1, numbers);
```

In this case matrix_create will allocate memory and do **memcpy** operation on the given pointer. If you want to create a matrix which has initially constant values you can use matrix_fill function:

```
matrix_t *out = matrix_create(uint16_t, 100, 100, 2);
uint16_t val[2] = {12, 16};
matrix_fill(out, &val);
```

The above code will generate a 100 x 100 matrix which has two dimensional inputs. The matrix_fill function will set all the first channel elements to 12 and the second channel elements to 16.

We can get the values from the matrix with various methods. In the simplest case if we know the data type of the matrix, we can obtain the data pointer and access the values via pointer just like standart c arrays.

```
matrix_t *out;
float *matrix_data = mdata(out);
for(size_t i = 0; i < volume(out); i++) {
    float val = matrix_data[i];
}
```

Or we can access the data element using the special function at as follows:

```
matrix_t *out;
for(size_t r = 0; i < rows(out); r++) {
    for(size_t c = 0; c < cols(out); c++) {
        float val = at(float, out, r,c);
    }
}
```

You can also use the matrix container with the custom defined structures. Here is an example of how to use matrix container with the custom structures.

```
struct person {
 uint32_t id;
 uint32_t birthday;
};
matrix_t *people = matrix_create(struct person, 5, 5);
struct person var = {1, 1990};
// we can set a matrix element using at
at(struct person, people, 0,0) = var;
// or we can use matrix_set
matrix_set(people, 0,0,0, &var);
 ...
matrix_free(&people);
```

The above code will generate a person structure and create a matrix container for that structure. Set the first element of the matrix and deallocate the memory when the all job is done.

Warning

> If the custom type has any dynamically allocated memory pointer an additional care must be taken in order to avoid memory leaks.

If the structure contains any pointer member than matrix_free will not free the memory allocated for the member and result in have memory leakage. Lets look at the problem with a similar example;

```
struct person {
 uint32_t id;
 char *name;
};
matrix_t *people = matrix_create(struct person, 5, 5);
struct person var = {1, strdup("any dynamically allocated memory inside the custom type will cause memory
       leak")};
// we can set a matrix element using at
at(struct person, people, 0,0) = var;
 ...
matrix_free(&people);
```

In this case the size of the structure is 40 byte but freeing 40 byte will only remove the memory of id and name variable but not the memory allocated for the name pointer. The only way to completely clean the object is first freeing the name pointer and than calling the matrix_free.

In order to make this process simple, IMLAB defines a function called matrix_destructor. This function is taking a destructor function for the created type and during matrix_free IMLAB first calls the given destructor function and then free up the memory.

```
struct person {
 uint32_t id;
 char *name;
};
// create a destructor for the person type
void person_destructor(void *elements, uint32_t plength) {
    struct *people = elements;
    for(size_t i=0; i < plength; i++) {
        free(people[i].name);
    }
}
matrix_t *people = matrix_create(struct person, 5, 5);
// set the destructor for the matrix
matrix_destructor(people, person_destructor);

struct person var = {1, strdup("in this case this memory will be released by the user defined destructor:
       person_destructor()")};
at(struct person, people, 0,0) = var;
...
matrix_free(&people);
```

See Also

matrix_create
matrix_resize
matrix_free
matrix_fill
matrix_load
matrix_save

4.6.2 Field Documentation

4.6.2.1 uint32_t matrix_t::_channels

4.6.2.2 uint32_t matrix_t::_cols

4.6.2.3 void* matrix_t::_data

4.6.2.4 uint32_t matrix_t::_rows

4.6.2.5 struct imlab_type_t* matrix_t::_type

The documentation for this struct was generated from the following file:

- E:/imlab_library/include/core.h

4.7 point_t Struct Reference

3D point type

```
#include <core.h>
```

Data Fields

- float x
- float y
- float z

4.7.1 Detailed Description

3D point type

4.7.2 Field Documentation

4.7.2.1 float point_t::x

4.7.2.2 float point_t::y

4.7.2.3 float point_t::z

The documentation for this struct was generated from the following file:

- E:/imlab_library/include/core.h

4.8 rectangle_t Struct Reference

Rectangle typed structure.

```
#include <core.h>
```

Data Fields

- int32_t x
- int32_t y
- int32_t width
- int32_t height
- float coefficient

4.8.1 Detailed Description

Rectangle typed structure.

4.8.2 Field Documentation

4.8.2.1 float rectangle_t::coefficient

4.8.2.2 int32_t rectangle_t::height

4.8.2.3 int32_t rectangle_t::width

4.8.2.4 int32_t rectangle_t::x

4.8.2.5 int32_t rectangle_t::y

The documentation for this struct was generated from the following file:

- E:/imlab_library/include/core.h

4.9 string_t Struct Reference

string_t is a string type to hold char pointers as string

```
#include <core.h>
```

Data Fields

- uint32_t _length
- uint32_t _capacity
- char * _data

4.9.1 Detailed Description

string_t is a string type to hold char pointers as string

4.9.2 Field Documentation

4.9.2.1 uint32_t string_t::_capacity

4.9.2.2 char* string_t::_data

4.9.2.3 uint32_t string_t::_length

The documentation for this struct was generated from the following file:

- E:/imlab_library/include/core.h

4.10 vector_t Struct Reference

vector_t is a vector container just like standart std vector in C++.

```
#include <core.h>
```

Data Fields

- struct imlab_type_t * _type
- uint32_t _capacity
- uint32_t _length
- void * _data

4.10.1 Detailed Description

vector_t is a vector container just like standart std vector in C++.

vector_t structure is a generic c-like array that can hold any type of variable in dynamically allocated memory. Just like the arrays, vector data is contigous in memory and can be accessed externally. The main advantage of the vector object onto the standart arrays is that the vector objects can dynamically grow or shrink. This type of needs can be accomplished via realloc and memcpy opeartions in standart C and as expected IMLAB is also doing these operations in clean and brief function calls and keeps everything about the vector in tiny and tidy vector_t structure.

Warning

> It is important to note that adding/deleting or changing an element or member of a vector should be done over the dedicated functions. Setting an element of the vector over data pointer does not change the length nor the capacity of the vector. This could cause memory overflow or data overwriting.

IMLAB defines the following functions for vector objects:

- length : returns the length of the vector
- capacity : returns the capacity of the vector
- vdata / data : returns the pointer to the data of the vector
- vector_create : constructor function and zero initializer
- vector_resize : change the size of the data holded in container
- vector_free : destructor functions
- vector_fill : set the struct with a constant value
- vector_permute : permute the entries of the vector with the given index list
- vector_push : push an instance to the vector, if there is no space, creates one
- vector_pop : get an index from the vector and remove it from the array
- vector_read : loads the vector struct from the hard drive (uncompress data)
- vector_write : saves the type struct to the hard drive (compress data)

Using a vector object is easy. Just decide the data type you need and call vector_create to create a new vector object. The data type can be any C data types, additional names declared in stdint.h or structs created by the user.

```
vector_t out = vector_create(uint32_t);
```

The above code will create a vector object and set the initial length to zero and capacity to one. If you want to specify an initial capacity to vector just call:

```
vector_t out = vector_create(float, 1024);
```

This call of the vector_create will create a floating point data pointer and allocate 1024*sizeof(float) byte memory on the memory. Since the vector object could be enlarged or shrinked, setting an initial capacity will increase the time efficincy for large vectors. It is also possible to create a vector from an existing one or from c pointers. In this case the third argument should be the pointer to the data:

```
double numbers[9] = {1,2,3, 4,5,6, 7,8,9};
vector_t out = vector_create(double, 9, numbers);
```

In this case vector_create will allocate memory and do **memcpy** operation on the given pointer. To create a vector which has initially constant values you can use vector_fill function:

```
vector_t out = vector_create(uint16_t, 100);
uint16_t val = 12;
vector_fill(&out, &val);
```

In order to add/remove element to vector vector_push/vector_pop can be used.

```
vector_t out = vector_create(float);
float values[12] = {0.0 0.1, 0.3, 0.5, 0.7, 0.9, 0.8, 0.6, 0.5, 0.4, 0.2, 0.0};
// in this loop the length of the vector will automatically increase up to 12.
for(size_t i = 0; i < 12; i++) {
    vector_push(&out, &values[i]);
}
```

We can get the values from the vector with various methods. In the simplest case if we know the data type of the vector, we can obtain the data pointer and access the values via pointer just like standart c arrays.

```
float *vector_data = vdata(out, 0);
for(size_t i = 0; i < length(out); i++) {
    float val = vector_data[i];
}
```

Or we can access the data elemnt using the special function at as follows:

```
for(size_t i = 0; i < length(out); i++) {
    float val = at(float, out, i);
}
```

You can also use the vector container with the custom defined structures. Here is an example of how to use vector container with the custom structures.

```
struct person {
 uint32_t id;
 uint32_t birthday;
};
vector_t *people = vector_create(struct person);
struct person var = {1, 1990};
vector_push(people, &var);
 ...
vector_free(&people);
```

The above code will generate a person structure and create a vector container for that structure. And push an instance of the structure into the people vector. And clear the vector after all the operations are done.

Warning

> If the custom type has any dynamically allocated memory pointer an additional care must be taken in order to avoid memory leaks.

If the structure contains any pointer member than matrix_free will not free the memory allocated for the member and result in memory leakage. Lets look at the problem with a similar example;

```
struct person {
 uint32_t id;
 char *name;
};
vector_t *people = vector_create(struct person, 50);
struct person var = {1, strdup("any dynamically allocated memory inside the custom type will cause memory
       leak")};
vector_push(people, &var);
 ...
vector_free(&people);
```

In this case the size of the structure is 40 byte but freeing 40 byte will only remove the memory of id and name variable but not the memory allocated for the name pointer. The only way to completely clean the object is first freeing the name pointer and than calling the vector_free.

In order to make this process simple, IMLAB defines a function called vector_destructor. This function is taking a destructor function for the created type and during vector_free IMLAB first calls the given destructor function and then free up the memory.

```
struct person {
 uint32_t id;
 char *name;
};
// create a destructor for the person type
void person_destructor(void *elements, uint32_t plength) {
    struct *people = elements;
    for(size_t i=0; i < plength; i++) {
        free(people[i].name);
    }
}
vector_t *people = vector_create(struct person, 50);
// set destructor for the contained type
vector_destructor(people,  person_destructor);

struct person var = {1, strdup("in this case this memory will be released by the user defined destructor:
       person_destructor()")};
vector_push(people, &var);
 ...
vector_free(&people);
```

See Also

vector_create
vector_resize
vector_free
vector_fill
vector_permute
vector_push
vector_pop
vector_read
vector_write

4.10.2 Field Documentation

4.10.2.1 uint32_t vector_t::_capacity

4.10.2.2 void* vector_t::_data

4.10.2.3 uint32_t vector_t::_length

4.10.2.4 struct imlab_type_t* vector_t::_type

The documentation for this struct was generated from the following file:

- E:/imlab_library/include/core.h

Chapter 5

File Documentation

5.1 mainpage/mainpage.dox File Reference

5.2 E:/imlab_library/include/alcore.h File Reference

```
#include <stdint.h>
#include "core.h"
```

Data Structures

- struct disjoint_set_t

Functions

- struct disjoint_set_t * disjoint_set_create (uint32_t length)
- uint32_t disjoint_set_find (struct disjoint_set_t *set, uint32_t idxP)
- return_t disjoint_set_union (struct disjoint_set_t *set, uint32_t idxP, uint32_t idxQ)
- uint32_t disjoint_set_enumerate (struct disjoint_set_t *set)
- void disjoint_set_free (struct disjoint_set_t **set)
- void disjoint_set_view (struct disjoint_set_t *set)

5.2.1 Function Documentation

5.2.1.1 struct disjoint_set_t* disjoint_set_create (uint32_t *length*)

Create a disjoint set data structure

Parameters

length	Length of the set

Returns

A disjoint set data structure

5.2.1.2 uint32_t disjoint_set_enumerate (struct disjoint_set_t * *set*)

Enumerate the roots of each element

Parameters

set	Disjoint set data structure

Returns

Number of unique root elements

5.2.1.3 uint32_t disjoint_set_find (struct disjoint_set_t * *set*, uint32_t *idxP*)

Fınds the root of the disjoint set element

Parameters

set	Disjoint set data structure
idxP	Index of the element

Returns

Root of the given element

5.2.1.4 void disjoint_set_free (struct disjoint_set_t ** *set*)

Free the memory allocated by the disjoint set data structure

Parameters

set	Disjoint set data structure

5.2.1.5 return_t disjoint_set_union (struct disjoint_set_t * *set*, uint32_t *idxP*, uint32_t *idxQ*)

Unions two element and updates the disjoint set tree

Parameters

set	Disjoint set data structure
idxP	Index of the first element
idxQ	Index of the second element

Returns

Success or relative error

5.2.1.6 void disjoint_set_view (struct disjoint_set_t * *set*)

Prints the current status of the disjoint set tree

Parameters

set	

5.3 E:/imlab_library/include/core.h File Reference

File containing example of doxygen usage for quick reference.

```
#include <stdio.h>
#include <stdint.h>
#include <stdarg.h>
#include <string.h>
#include "core_macros.h"
```

Data Structures

- struct vector_t

  vector_t is a vector container just like standart std vector in C++.
- struct matrix_t

  matrix_t is a n-channel 2D matrix container.
- struct string_t

  string_t is a string type to hold char pointers as string
- struct rectangle_t

  Rectangle typed structure.
- struct point_t

  3D point type

Macros

- #define IM_VERBOSE_ERROR 1
- #define IM_VERBOSE_WARNING 1
- #define IM_VERBOSE_SUCCESS 1
- #define IMLAB_USE_OPENMP 1
- #define idx(...)

  Return the index of the data at the given position.
- #define elemidx(...)

  Return the index of the given position in a byte array.
- #define elemsize(_var)

  Return the size of the element hold in the data pointer.
- #define at(_type,...)

  Returns the data value at the given position for the given imlab object.
- #define data(_type,...)

  Returns the data pointer for the given imlab container.
- #define typeof(_var)

  Return the type information of the given IMLAB container.
- #define typeid(_var)

  Return the type id of the given IMLAB container.
- #define typename(_var)

  Return the type of the given container as string.
- #define type(type_name)

  Create a new IMLAB container compatible type from the given type name.
- #define message(cond,...) print_message_func(cond, __LINE__, __func__, __VA_ARGS__)
- #define array_create(...)

  Create a new array with the specified input paramaters. This macro definition calls the array_create function with the correct values of arguments. This macro overloaded with respect to number of arguments. The actual function is implemented to support one to four input arguments. The macro implementation could take up to five arguments as follows:
- #define vector_create(...)

Create a new vector with the specified input paramaters. This macro definition calls the vector_create function with the appropriate number of arguments. This macro overloaded with respect to number of arguments. The actual function takes three input arguments so the macro implementation could take up to three arguments as follows:

- #define vector_convert(_t1, _t2, _in)

  This method converts the input vector (t1 typed) into t2 typed vector by just casting values.
- #define matrix_create(...)

  Creates a new matrix element with the specified paramaters. This macro overloaded with respect to number of arguments. The actual function takes five input arguments so the macro implementation could take up to five arguments as follows:
- #define matrix_convert(_t1, _t2, _in)

  This method converts the input matrix (t1 typed) into t2 typed matrix by just casting values.

IMLAB Type Comparators

Returns true if the asked question is true for matrix/vector

- #define is_8s(var)

  return true if the given matrix or vector holds NULL data type
- #define is_8u(var)

  return true if the given matrix or vector holds uint8_t data type
- #define is_16s(var)

  return true if the given matrix or vector holds int16_t data type
- #define is_16u(var)

  return true if the given matrix or vector holds uint16_t data type
- #define is_32s(var)

  return true if the given matrix or vector holds int32_t data type
- #define is_32u(var)

  return true if the given matrix or vector holds uint32_t data type
- #define is_32f(var)

  return true if the given matrix or vector holds float data type
- #define is_64f(var)

  return true if the given matrix or vector holds double data type
- #define is_json_array(var)

  return true if the given matrix or vector holds a JSON array
- #define is_json_object(var)

  return true if the given matrix or vector holds a JSON object
- #define is_image(var)

  return true if the given matrix or vector is image
- #define is_integer(var)

  return true if the given matrix or vector is integer
- #define is_numeric(var)

  return true if the given matrix or vector is floating point number
- #define is_sametype(var1, var2)

  return true if the given two matrice or vectors have the same type

Enumerations

- enum return_t {
  ERROR_NOT_IMAGE = -7, ERROR_NULL_TYPE = -6, ERROR_TYPE_MISMATCH = -5, ERROR_DIMENSION_MISMATCH = -4,
  ERROR_OUT_OF_MEMORY = -3, ERROR_UNABLE_TO_OPEN = -2, ERROR = -1, SUCCESS = 0,
  WARNING = 1, WARNING_NOT_SUPPORTED = 2, WARNING_NOTHING_DONE = 3 }

  IMLAB return type Return type for most of the IMLAB function. Returns a negative integer when an error occurs. Than this error can be categorized using the value of the return. Return a positive integer when everything done without any problem.

Functions

- int print_message_func (return_t cond, int line, const char ∗func, const char ∗fmt,...)

    Print colored and formatted debug message into the stderr. This function called via message macro.
- uint32_t array_size (void ∗head, uint32_t dim)
- void array_free (void ∗head)

    Free the memory allocated by the given array and set the pointer to NULL.
- vector_t ∗ vector_null ()
- return_t vector_resize (vector_t ∗var, uint32_t _capacity)
- return_t vector_push (vector_t ∗var, void ∗element)
- return_t vector_pop (vector_t ∗var, void ∗element)
- void vector_set (vector_t ∗var, uint32_t idx, void ∗value)
- void vector_get (vector_t ∗var, uint32_t idx, void ∗value)
- return_t vector_permute (vector_t ∗in, uint32_t ∗index_list)
- void vector_free (vector_t ∗∗var)
- void vector_destructor (vector_t ∗in, void(∗func)(void ∗, uint32_t))
- return_t vector_fill (vector_t ∗var, void ∗value)
- return_t vector_write (vector_t ∗src, const char ∗filename)
- vector_t ∗ vector_read (const char ∗filename)
- return_t vector_unique (vector_t ∗src, vector_t ∗uniques, vector_t ∗unique_idx)
- void vector_view (vector_t ∗in)
- void ∗ vdata (vector_t ∗this, uint32_t idx)
- uint32_t length (vector_t ∗this)
- uint32_t capacity (vector_t ∗this)
- struct imlab_type_t ∗ vector_type (vector_t ∗this)
- matrix_t ∗ matrix_null ()
- void matrix_set (matrix_t ∗var, uint32_t rows, uint32_t cols, uint32_t channels, void ∗value)

    Set the value of a matrix element to the given variable.
- void matrix_get (matrix_t ∗var, uint32_t rows, uint32_t cols, uint32_t channels, void ∗value)

    Set the given variable to the value of a matrix element in var(rows,cols,channels) position.
- return_t matrix_resize (matrix_t ∗var, uint32_t nrows, uint32_t ncols, uint32_t nchannels)
- vector_t ∗ matrix2vector (matrix_t ∗input, uint8_t order)
- void matrix_free (matrix_t ∗∗var)
- void matrix_destructor (matrix_t ∗in, void(∗func)(void ∗, uint32_t))
- return_t matrix_fill (matrix_t ∗out, void ∗value)
- return_t matrix_copy (matrix_t ∗src, matrix_t ∗dst)
- return_t matrix_write (matrix_t ∗src, const char ∗filename)
- void matrix_view (matrix_t ∗in)
- matrix_t ∗ matrix_read (const char ∗filename)
- struct imlab_type_t ∗ matrix_type (matrix_t ∗this)
- void ∗ mdata (matrix_t ∗this, uint32_t idx)
- uint32_t width (matrix_t ∗this)
- uint32_t height (matrix_t ∗this)
- uint32_t rows (matrix_t ∗this)
- uint32_t cols (matrix_t ∗this)
- uint32_t channels (matrix_t ∗this)
- uint32_t volume (matrix_t ∗this)
- string_t string (char ∗cstr)
- char ∗ c_str (string_t str)
- return_t string_append (char ∗cstr, string_t ∗str)
- return_t string_merge (string_t ∗out_str, string_t ∗in_str)
- void string_destruct (void ∗in, uint32_t length)
- return_t string_printf (string_t ∗str, const char ∗format,...)
- return_t string_restart (string_t ∗str)

- struct rectangle_t rectangle (int32_t x, int32_t y, int32_t width, int32_t height, float coefficient)
- float rectangle_overlap (struct rectangle_t r1, struct rectangle_t r2, uint8_t mode)
- vector_t * rectangle_merge (vector_t *rect, float threshold, uint8_t method)
- struct point_t point (float x, float y, float z)

    Create a new point with the given parameters.
- float point_distance (struct point_t p1, struct point_t p2)

    Computes the distance between the given two points.

5.3.1 Detailed Description

File containing example of doxygen usage for quick reference.

Author

My Self

Date

9 Sep 2012 Here typically goes a more extensive explanation of what the header defines. Doxygens tags are words preceeded by either a backslash \ or by an at symbol @.

See Also

http://www.stack.nl/~dimitri/doxygen/docblocks.html
http://www.stack.nl/~dimitri/doxygen/commands.html

5.3.2 Macro Definition Documentation

5.3.2.1 #define array_create(...)

Create a new array with the specified input paramaters. This macro definition calls the array_create function with the correct values of arguments. This macro overloaded with respect to number of arguments. The actual function is implemented to support one to four input arguments. The macro implementation could take up to five arguments as follows:

```
array_create(<type>, dim1length, dim2length = 0, dim3length = 0, dim4length = 0)
```

In the simplest case the function could take the only type of the array and length of the array. This type could be any c types defined in stdint.h or struct created by the user.

```
uint32_t i;
uint32_t *out = array_create(uint32_t, 256);
for(i = 0; i < 256; i++) {
    out[i] = i * 3;
}
array_free(out);
```

The above code will create an array object with the given length and initialize it with zeros. Than it can be used just as regular c pointers. The only important remark is to free the pointer allocated by array_create using the array_free. If you want to create multi dimensional array, call the function as follows:

```
float **out = array_create(float, 1024, 100);
for(i = 0; i < array_size(out,0); i++) {
    for(j = 0; j < array_size(out,1); j++) {
        out[i][j] = (i * j) / 1024;
    }
}
array_free(out);
```

This call of the array_create will create a 2 dimensional floating point data pointer and allocate 1024*100*sizeof(float) byte memory on the memory. It can be used as regular two dimensional arrays.

Parameters

<type>	Content type of the array data. It could be any type char,uint32_t, float or user defined structures.
dimXlength	Length of the output array in the Xth dimension.

5.3.2.2 #define at(_type, ...)

Returns the data value at the given position for the given imlab object.

It is overloaded with the number of input arguments. In the simplest case data takes two argument which is the type and imlab struct and returns the dataa at the zero index of the given variable.

```
matrix_t *matf = matrix_create(float, 100, 100);
float mdata = at(float, matf); // mdata points to matf->_data(0,0,0)
```

at macro can also be called with a starting index as follows:

```
vector_t veci = vector_create(uint32_t, 100);
uint32_t vdata = at(uint32_t, veci, 5); // vdata points to the veci->_data(5)
```

This macro can also be used to access a specific position of a matrix:

```
matrix_t mati = matrix_create(uint32_t, 100, 100, 3);
uint32_t m1data = at(uint32_t, mati, 5); // m1data points to the mati->_data(0,5,0)
uint32_t m2data = at(uint32_t, mati, 5, 7); // m2data points to the mati->_data(5,7,0)
uint32_t m3data = at(uint32_t, mati, 5, 7, 2); // m3data points to the mati->_data(5,7,2)
```

5.3.2.3 #define data(_type, ...)

Returns the data pointer for the given imlab container.

It is overloaded with the number of input arguments. In the simplest case data takes two argument which are the type and imlab struct and returns the pointer to the data.

```
matrix_t matf = matrix_create(float, 100, 100);
float *matf_data = data(float, matf); // mdata points to &(matf->_data(0,0,0))
```

data macro can also be called with a starting index as follows:

```
vector_t veci = vector_create(uint32_t, 100);
uint32_t *v1data = data(uint32_t, veci, 5); // v1data points to the &(veci->_data(5))
```

This macro can also be used to access a specific position of a matrix:

```
matrix_t mati = matrix_create(uint32_t, 100, 100, 3);
uint32_t *m1data = data(uint32_t, mati, 5); // m1data points to the &(mati->_data(0,5,0))
uint32_t *m2data = data(uint32_t, mati, 5, 7); // m2data points to the &(mati->_data(5,7,0))
uint32_t *m3data = data(uint32_t, mati, 5, 7, 2); // m3data points to the &(mati->_data(5,7,2))
```

5.3.2.4 #define elemidx(...)

Return the index of the given position in a byte array.

```
matrix_t *matA = matrix_create(float, 5,5);
matrix_t *vecA = vector_create(uint16_t, 5);

uint32_t i1 = elemidx(matA, 0,1); // 4
uint32_t i2 = elemidx(matA, 0,2); // 8
uint32_t i3 = elemidx(matA, 1,1); // 24
uint32_t i4 = elemidx(matA, 1,2); // 28

uint32_t i5 = elemidx(vecA, 1); // 2
uint32_t i6 = elemidx(vecA, 2); // 4
```

5.3.2.5 #define elemsize(_var)

Return the size of the element hold in the data pointer.

```
matrix_t *matA = matrix_create(float, 5,5);
matrix_t *vecA = vector_create(uint16_t, 5);
// get the element size of the container data
uint32_t i1 = elemsize(matA); // 4
uint32_t i2 = elemsize(vecA); // 2
```

5.3.2.6 #define idx(...)

Return the index of the data at the given position.

```
matrix_t *matA = matrix_create(float, 5,5);
matrix_t *vecA = vector_create(uint16_t, 5);

uint32_t i1 = idx(matA, 0,1); // 1
uint32_t i2 = idx(matA, 0,2); // 2
uint32_t i3 = idx(matA, 1,1); // 6
uint32_t i4 = idx(matA, 1,2); // 7

uint32_t i5 = idx(vecA, 1); // 1
uint32_t i6 = idx(vecA, 2); // 2
```

5.3.2.7 #define IM_VERBOSE_ERROR 1

5.3.2.8 #define IM_VERBOSE_SUCCESS 1

5.3.2.9 #define IM_VERBOSE_WARNING 1

5.3.2.10 #define IMLAB_USE_OPENMP 1

5.3.2.11 #define is_16s(var)

return true if the given matrix or vector holds int16_t data type

5.3.2.12 #define is_16u(var)

return true if the given matrix or vector holds uint16_t data type

5.3.2.13 #define is_32f(var)

return true if the given matrix or vector holds float data type

5.3.2.14 #define is_32s(var)

return true if the given matrix or vector holds int32_t data type

5.3.2.15 #define is_32u(var)

return true if the given matrix or vector holds uint32_t data type

5.3.2.16 #define is_64f(var)

return true if the given matrix or vector holds double data type

5.3.2.17 #define is_8s(*var*)

return true if the given matrix or vector holds NULL data type

return true if the given matrix or vector holds int8_t data type

5.3.2.18 #define is_8u(*var*)

return true if the given matrix or vector holds uint8_t data type

5.3.2.19 #define is_image(*var*)

return true if the given matrix or vector is image

5.3.2.20 #define is_integer(*var*)

return true if the given matrix or vector is integer

5.3.2.21 #define is_json_array(*var*)

return true if the given matrix or vector holds a JSON array

5.3.2.22 #define is_json_object(*var*)

return true if the given matrix or vector holds a JSON object

5.3.2.23 #define is_numeric(*var*)

return true if the given matrix or vector is floating point number

5.3.2.24 #define is_sametype(*var1, var2*)

return true if the given two matrice or vectors have the same type

5.3.2.25 #define matrix_convert(*_t1, _t2, _in*)

This method converts the input matrix (t1 typed) into t2 typed matrix by just casting values.

```
float data[4] = {1.1, 2.2, 3.3, 4.4};
matrix_t *matA = matrix_create(float, 1,4,1, data);
// print the type name
printf("Type of matA: %s\n", typename(matA));
// print the values
printf("[%3.2f %3.2f %3.2f %3.2f]\n", at(float, matA, 0), at(float, matA, 1),
      at(float, matA, 2), at(float, matA, 3));
// convert the values into int type
matrix_convert(float, int32_t, matA);
// print the type name
printf("Type of matA: %s\n", typename(matA));
// print the values
printf("[%d %d %d %d]\n", at(int32_t, matA, 0), at(int32_t, matA, 1), at(int32_t, matA, 2),
      at(int32_t, matA, 3));

// the above code will print
// Type of matA: float
// [1.10 2.20 3.30 4.40]
// Type of matA: int32_t
// [1 2 3 4]
```

Warning

Note that changing the access type of at function does not change the type of the matrix or the data in the memory. In at or data, type variable is used to determine the size of one element and jump to the correct index.

5.3.2.26 #define matrix_create(...)

Creates a new matrix element with the specified paramaters. This macro overloaded with respect to number of arguments. The actual function takes five input arguments so the macro implementation could take up to five arguments as follows:

```
matrix_create(<type>, uint32_t rows = 0, uint32_t cols = 0, uint32_t channels = 0, void *pointer = NULL)
```

Missing arguments are replaced with the default values. In the simplest case the function could take the only type of the matrix. This type could be any c types defined in stdint.h or struct created by the user.

```
matrix_t *out = matrix_create(uint32_t);
```

The above code will create a matrix object and set the initial size to zero. If you want to specify the size of the matrix at the construction call:

```
matrix_t *out = matrix_create(float, 1024, 1024);
```

This call of the matrix_create will create a floating point data pointer and allocate 1024∗1024∗sizeof(float) byte memory on the memory. It is also possible to create a matrix from an existing one or from c pointers. In this case the third argument should be the pointer to the data:

```
double numbers[9] = {1,2,3, 4,5,6, 7,8,9};
// create a new matrix from the given C array
matrix_t *out = matrix_create(double, 3,3,1, numbers);
// create a new matrix with the type and size of the out
matrix_t *out_copy = matrix_create(out, NULL);
// create a new matrix with the type, size and data of the out
matrix_t *out_clone = matrix_create(out, data(out));
```

In this case matrix_create will allocate memory and do **memcpy** operation on the given pointer.

Parameters

<type>	Content type of the matrix data. It could be any type char,uint32_t, float or user defined structures.
rows	Number of rows (height) of the output matrix.
cols	Number of columns (width) of the output matrix.
channels	Number of channels of the output matrix.
pointer	A pointer to be set to the matrix data

5.3.2.27 #define message(*cond,* ...) **print_message_func**(cond, __LINE__, __func__, __VA_ARGS__)

5.3.2.28 #define type(*type_name*)

Create a new IMLAB container compatible type from the given type name.

5.3.2.29 #define typeid(*_var*)

Return the type id of the given IMLAB container.

5.3.2.30 #define typename(_var)

Return the type of the given container as string.

5.3.2.31 #define typeof(_var)

Return the type information of the given IMLAB container.

5.3.2.32 #define vector_convert(_t1, _t2, _in)

This method converts the input vector (t1 typed) into t2 typed vector by just casting values.

```
float data[4] = {1.1, 2.2, 3.3, 4.4};
matrix_t *vecA = vector_create(float, 4, data);
// print the type name
printf("Type of vecA: %s\n", typename(vecA));
// print the values
printf("[%3.2f %3.2f %3.2f %3.2f]\n", at(float, vecA, 0), at(float, vecA, 1),
      at(float, vecA, 2), at(float, vecA, 3));
// convert the values into int type
vector_convert(float, int32_t, vecA);
// print the type name
printf("Type of vecA: %s\n", typename(matA));
// print the values
printf("[%d %d %d %d]\n", at(int32_t, vecA, 0), at(int32_t, vecA, 1), at(int32_t, vecA, 2),
      at(int32_t, vecA, 3));

// the above code will print
// Type of vecA: float
// [1.10 2.20 3.30 4.40]
// Type of vecA: int32_t
// [1 2 3 4]
```

Warning

> Note that changing the access type of at function does not change the type of the matrix or the data in the memory. In at or data, type variable is used to determine the size of one element and jump to the correct index.

5.3.2.33 #define vector_create(...)

Create a new vector with the specified input paramaters. This macro definition calls the vector_create function with the appropriate number of arguments. This macro overloaded with respect to number of arguments. The actual function takes three input arguments so the macro implementation could take up to three arguments as follows:

```
vector_create(<type>, uint32_t capacity = 1, void *pointer = NULL)
```

Missing arguments are replaced with the default values. In the simplest case the function could take the only type of the vector. This type could be any c types defined in stdint.h or struct created by the user.

```
vector_t *out = vector_create(uint32_t);
```

The above code will create a vector object and set the initial length to zero and capacity to one. If you want to specify an initial capacity to vector just call:

```
vector_t *out = vector_create(float, 1024);
```

This call of the vector_create will create a floating point data pointer and allocate 1024*sizeof(float) byte memory on the memory. Since the vector object could be enlarged or shrinked, setting an initial capacity will increase the time efficincy for large vectors. It is also possible to create a vector from c pointers. In this case the third argument should be the pointer to the data:

```
double numbers[9] = {1,2,3, 4,5,6, 7,8,9};
vector_t *out = vector_create(double, 9, numbers);
```

In this case vector_create will allocate memory and do **memcpy** operation on the given pointer.

Parameters

<type>	Content type of the vector data. It could be any type char,uint32_t, float or user defined structures.
capacity	Capacity of the output vector. If you dont know the exact size of the vector at the programming stage just write the expected length of the vector. IMLAB automatically increase the size if the data exceeds the vector length (This is only true if you use vector_push for the data insertion).
pointer	A pointer to the vector data. If this parameter is send to the function imlab will allocate a new memory and copies the send pointer into the container.

Returns

A vector_t object.

5.3.3 Enumeration Type Documentation

5.3.3.1 enum **return_t**

IMLAB return type Return type for most of the IMLAB function. Returns a negative integer when an error occurs. Than this error can be categorized using the value of the return. Return a positive integer when everything done without any problem.

Enumerator

ERROR_NOT_IMAGE
ERROR_NULL_TYPE
ERROR_TYPE_MISMATCH
ERROR_DIMENSION_MISMATCH
ERROR_OUT_OF_MEMORY
ERROR_UNABLE_TO_OPEN
ERROR
SUCCESS
WARNING
WARNING_NOT_SUPPORTED
WARNING_NOTHING_DONE

5.3.4 Function Documentation

5.3.4.1 void array_free (void ∗ *head*)

Free the memory allocated by the given array and set the pointer to NULL.

Parameters

head	Input array to be deleted.

Remarks

Input array must be allocated by array_create

5.3.4.2 uint32_t array_size (void ∗ *head,* uint32_t *dim*)

Return the size of the array in the given dimension. If the array is NULL or dim is higher than 4, this function returns 0.

Parameters

head	Input array allocated by array_create
dim	Dimension of the input array

Returns

Length of the array in the given dimension

5.3.4.3 char∗ c_str (string_t *str*)

Return the C-compatible string of the string object

Parameters

str	A string_t object

Returns

C-compatible string

5.3.4.4 uint32_t capacity (vector_t ∗ *this*) [inline]

Returns the capacity of the vector

Parameters

this	Input vector

Returns

Capacity of the input vector

5.3.4.5 uint32_t channels (matrix_t ∗ *this*) [inline]

Returns the channels of the matrix

Parameters

this	Input matrix

Returns

Channels of the input matrix

5.3.4.6 uint32_t cols (matrix_t ∗ *this*) [inline]

Returns the columns of the matrix

Parameters

this	Input matrix

Returns

Columns of the input matrix

5.3.4.7 uint32_t height (matrix_t * this) [inline]

Returns the height of the matrix

Parameters

this	Input matrix

Returns

Height of the input matrix

5.3.4.8 uint32_t length (vector_t * *this*) [inline]

Returns the length of the vector

Parameters

this	Input vector

Returns

Length of the input vector

5.3.4.9 vector_t* matrix2vector (matrix_t * *input*, uint8_t *order*)

Convert the matrix into a vector

Parameters

input	Input matrix
order	Row first or coloumn first conversion

Returns

Vector filled with matrix element

5.3.4.10 return_t matrix_copy (matrix_t * *src*, matrix_t * *dst*)

Copy source matrix into the destination matrix, resize the destination if needed

Parameters

src	Source matrix to be copied
dst	Destination matrix (same type with the source)

Returns

Success or relative error

5.3.4.11 void matrix_destructor (matrix_t * *in*, void(*)(void *, uint32_t) *func*)

Sets a destructor for the data holded in the matrix container

Parameters

in	Input matrix
func	Destructor function to be set

5.3.4.12 return_t matrix_fill (matrix_t * *out,* void * *value*)

Fill a matrix with the desired value. If there are multiple channels(channels != 1) than instead of sending single variable, send array of data which length is equal to channels double array[] = {255,0,128}; matrix_fill(color_image, array) by this way, _fill will copy 255 to the first, 0 to second and 128 to the third channel of the image

Parameters

_var	Input matrix to be filled.
_value	Data to be written to each cell of the input matrix.

5.3.4.13 void matrix_free (matrix_t ** *var*)

Release the allocated memory for the matrix object.

Parameters

_var	Input matrix to be freed.

5.3.4.14 void matrix_get (matrix_t * *var,* uint32_t *rows,* uint32_t *cols,* uint32_t *channels,* void * *value*)

Set the given variable to the value of a matrix element in var(rows,cols,channels) position.

Parameters

var	Input matrix
rows	Row of the matrix element to be get
cols	Column of the matrix element to be get
channels	Channels channel of the matrix element to be get
value	Address of the variable to be get into

5.3.4.15 matrix_t* matrix_null ( )

Create a NULL matrix

Returns

5.3.4.16 matrix_t* matrix_read (const char * *filename*)

Read the matrix from the file

Parameters

filename	Filename of the matrix
dst	Non allocated matrix container for the destination

Returns

Success or relative error

5.3.4.17 return_t matrix_resize (matrix_t * *var*, uint32_t *nrows*, uint32_t *ncols*, uint32_t *nchannels*)

Resize the input matrix to the specified size. This function is used most of the functions in order to create output matrix.

Parameters

_var	Input matrix to be resized.
_nrows	New rows of the input matrix.
_ncols	New columns of the input matrix.
_nchannels	New channels of the input matrix.

5.3.4.18 void matrix_set (matrix_t * *var*, uint32_t *rows*, uint32_t *cols*, uint32_t *channels*, void * *value*)

Set the value of a matrix element to the given variable.

Parameters

var	Input matrix
rows	Row of the matrix element to be set
cols	Column of the matrix element to be set
channels	Channel of the matrix element to be set
value	Address of the variable to be set

5.3.4.19 struct imlab_type_t* matrix_type (matrix_t * *this*)

Returns the type of the matrix

Parameters

this	Input matrix

Returns

Type information of the matrix

5.3.4.20 void matrix_view (matrix_t * *in*)

Prints the matrix information into the stdout

Parameters

in	Input matrix to be viewed

5.3.4.21 return_t matrix_write (matrix_t * *src*, const char * *filename*)

Save the input matrix to the disk

Parameters

src	Input matrix to be saved
filename	Filename for the output file

Returns

Success or relative error

5.3.4.22 void* mdata (matrix_t * *this*, uint32_t *idx*) [inline]

Return the data pointer of the matrix

Parameters

this	Input matrix

Returns

Pointer to the data of the matrix

5.3.4.23 struct point_t point (float *x*, float *y*, float *z*)

Create a new point with the given parameters.

Parameters

x	x position of the point
y	y position of the point
z	z position of the point

Returns

A point object created in stack

5.3.4.24 float point_distance (struct point_t *p1*, struct point_t *p2*)

Computes the distance between the given two points.

Parameters

p1	First point
p2	Second point

Returns

Euclidean distance between the points

5.3.4.25 int print_message_func (return_t *cond*, int *line*, const char * *func*, const char * *fmt*, ...)

Print colored and formatted debug message into the stderr. This function called via message macro.

5.3.4.26 struct rectangle_t rectangle (int32_t *x*, int32_t *y*, int32_t *width*, int32_t *height*, float *coefficient*)

Create a new rectangle with the given parameters

Parameters

x	x position of the top-left corner
y	x position of the top-left corner
width	Width of the rectangle
height	Height of the rectangle
coefficient	A floating point number assign to the rectangle (it could be detection score, color, opacity vs)

Returns

A rectangle object created in stack

5.3.4.27 vector_t* rectangle_merge (vector_t * *rect*, float *threshold*, uint8_t *method*)

5.3.4.28 float rectangle_overlap (struct rectangle_t *r1*, struct rectangle_t *r2*, uint8_t *mode*)

Computes the overlapping area between the two rectangle objects.

Parameters

box1	First rectangle object
box2	Second rectangle object
mode	0 for overlapping area between the two rectangle and 1 for normalized intersection area

Returns

Area of the intersection

5.3.4.29 uint32_t rows (matrix_t * this) [inline]

Returns the rows of the matrix

Parameters

this	Input matrix

Returns

Rows of the input matrix

5.3.4.30 string_t string (char * cstr)

Create a string from the given char pointer

Parameters

cstr	Input C string (char*)

Returns

string_t object

5.3.4.31 return_t string_append (char * cstr, string_t * str)

Append the given C string into the string_t variable

Parameters

cstr	Input C string
str	Output string_t object

Returns

Success or relative error

5.3.4.32 void string_destruct (void * in, uint32_t length)

Destructor of string_t *data array

Parameters

in	Input string_t *array
length	Length of the pointer

5.3.4.33 return_t string_merge (string_t * *out_str*, string_t * *in_str*)

Merges the given two string_t object and writes the output string into the first element

Parameters

out_str	First string element to be merged and also the output of the merge operation
in_str	Second string object to be merged with the first one

Returns

Success or relative error

5.3.4.34 return_t string_printf (string_t * *str*, const char * *format*, ...)

Appends the given format string into the

Parameters

str.	
str	Output string
format	Format string to be written
...	

Returns

```
// first create an empty string
string_t stream = string("");
// fill the string with the given text
string_printf(&stream, "hello");
string_printf(&stream, "%c", 32);
string_printf(&stream, "IML@B");
// print the string
printf("%s\n", c_str(stream));

// the above code will print
// hello IML@B
```

5.3.4.35 return_t string_restart (string_t * *str*)

5.3.4.36 void* vdata (vector_t * *this*, uint32_t *idx*) [inline]

Returns the data pointer of the vector

Parameters

this	Input vector

Returns

Pointer to the start of the vector data

5.3.4.37 void vector_destructor (vector_t * *in*, void(*)(void *, uint32_t) *func*)

Sets a destructor for the data holded in the vector container

Parameters

in	Input vector
func	Destructor function to be set

5.3.4.38 return_t vector_fill (vector_t * *var*, void * *value*)

Parameters

var	Input vector to be filled.
value	Pointer to the variable to be filled into the vector element.

Returns

5.3.4.39 void vector_free (vector_t ** *var*)

Release the allocated memory for the vector object.

Parameters

_var	Input vector to be freed.

5.3.4.40 void vector_get (vector_t * *var*, uint32_t *idx*, void * *value*)

Gets the idx'th element of the vector to the given variable. This is not the correct way to extract an element from the vector. Use vector_pop for extracting an element from the vector. This function is intented to be used internally or in loops so in order to be fast it doesnt control the idx and it will create a serious bug if you try to get an element at any index larger than the length of the vector and it will create a buffer overflow if index is larger than the capacity of the vector.

Parameters

var	Input vector.
idx	Index of the vector element to be set.
value	Address of the data to be set.

5.3.4.41 vector_t* vector_null ( )

Create a NULL vector. This vector will hold sizeof(vector_t) bytes of memory and should be used for all vector pointers that are not created at the declaration.

Returns

5.3.4.42 return_t vector_permute (vector_t * *in*, uint32_t * *index_list*)

Permute the entries of the vector with the given index list.

Parameters

_var	Input vector to be permuted.
_idx	Index list for the permuted vector.

5.3.4.43 return_t vector_pop (vector_t * *var,* void * *element*)

Pops an element from the end of the vector and deletes the last element of the vector. This method does not change the capacity of the vector. You can manually resize the vector if you want to use less memory.

Parameters

_var	Input vector.
_element	Address of an element to be popped into.

5.3.4.44 return_t vector_push (vector_t * *var,* void * *element*)

Pushes an element into the end of the vector. If the length of the vector exceeds the vector length it automatically allocate memory for the new data.

Parameters

_var	Input vector.
_element	Address of the data to be pushed.

5.3.4.45 vector_t* vector_read (const char * *filename*)

Read the input vector from the given file.

Parameters

filename	Input file name with an appropriate extension.

Returns

Read vector from the file

5.3.4.46 return_t vector_resize (vector_t * *var,* uint32_t *_capacity*)

Resize the input vector to the specified size.

Parameters

var	Input vector to be resized.
ncapacity	New capacity of the input vector. If the new length is larger than the current length, the contents of the vector is protected. If the new length is smaller than the current length, the #(new length) of the input vector is protected.

5.3.4.47 void vector_set (vector_t * *var,* uint32_t *idx,* void * *value*)

Sets the idx'th element of the vector to the given value. This is not the correct way to insert an element to the vector. Use vector_push for adding a new element to the vector. This function is intented to be used internally or in loops so in order to be fast it doesnt control the idx and it will create a serious bug if you try to insert an element at any index larger than the length of the vector and it will create a buffer overflow if index is larger than the capacity of the vector.

Parameters

var	Input vector.
idx	Index of the vector element to be set.
value	Address of the data to be set.

5.3.4.48 struct imlab_type_t* vector_type (vector_t * *this*)

Returns the type of the vector

Parameters

this	Input vector

Returns

Type information of the input vector

5.3.4.49 return_t vector_unique (vector_t * *src*, vector_t * *uniques*, vector_t * *unique_idx*)

Return the unique elements of the vector

Parameters

src	Input vector
uniques	Unique elements created with the same type of the source
unique_idx	Index of the unique elements

Returns

5.3.4.50 void vector_view (vector_t * *in*)

Print the properties of the vector container

Parameters

in	Input vector to be viewed

5.3.4.51 return_t vector_write (vector_t * *src*, const char * *filename*)

Save the input vector to the given file.

Parameters

filename	Output file name with an appropriate extension.
src	Input vector structure to be saved. The default option (0) saves the file without any compression.

5.3.4.52 uint32_t volume (matrix_t * *this*) [inline]

Returns the volume of the matrix

Parameters

this	Input matrix

Returns

Volume (Rows ∗ Cols ∗ Channels) of the input matrix

5.3.4.53 uint32_t width (matrix_t ∗ *this*) [inline]

Returns the width of the matrix

Parameters

this	Input matrix

Returns

Width of the input matrix

5.4 E:/imlab_library/include/core_macros.h File Reference

Macros

- #define swap(_type, a, b)
- #define clamp(x, lower, higher)
- #define equal(a, b, eps)
- #define roundup(a, m)
- #define remedy(a, m)
- #define map(_in, _imin, _imax, _omin, _omax)
- #define deg2rad(angle)
- #define rad2deg(angle)
- #define square(x)
- #define stringify(txt)
- #define va_nargs(...)
- #define call(func,...)
- #define arg(_i,...)
- #define cat(...)
- #define max(...)
- #define min(...)
- #define in_range(_var, _min, _max) ((_var) >= (_min) && (_var) <= (_max))

5.4.1 Macro Definition Documentation

5.4.1.1 #define arg(*_i, ...*)

Get the argument _i in a variable length argument list (up to 9 arguments)

5.4.1.2 #define call(*func, ...*)

Call the given function_(#number of arguments) with the given arguments

5.4.1.3 #define cat(...)

Concanate the given arguments (up to 9 arguments)

```
cat(1.2345);  // 1.2345
cat(1,2);     // 12
cat(a,b,c,d); // abcd
```

5.4.1.4 #define clamp(*x,* *lower,* *higher*)

Clamp the given variable x, between the lower and higher limits

```
double a = 1.4, b = 1.6, c = b-a;
clamp(a, 0.5,1.5); // a : 1.4 (unchanged)
clamp(b, 0.5,1.5); // b : 1.5 (clamp to higher limit)
clamp(c, 0.5, 1.5) // c : 0.5 (clamp to lower limit)
```

5.4.1.5 #define deg2rad(*angle*)

Convert degrees to radians

5.4.1.6 #define equal(*a,* *b,* *eps*)

Compare the given variable a and b, and return true if they are in eps neighbourhood

```
double a = 1.4, b = 1.6;
equal(a, b, 0.1); // false
equal(a, b, 0.3); // true
```

5.4.1.7 #define in_range(*_var,* *_min,* *_max*) ((_var) >= (_min) && (_var) <= (_max))

5.4.1.8 #define map(*_in,* *_imin,* *_imax,* *_omin,* *_omax*)

Map input value which in [_imin, _imax] range to value between [_omin, _omax]

```
double angle = 90;
map(angle, -180,180, -pi,pi); // returns pi/4
```

5.4.1.9 #define max(...)

Max macro for different number of arguments (up to 9 arguments)

```
max(1,2);      // 2
max(min(7,8),11,12); // 12
```

5.4.1.10 #define min(...)

Min macro for different number of arguments (up to 9 arguments)

```
min(1.2345);  // 1.2345
min(a,b,c,d); // compares the values in (a,b,c,d) and return the minimum
```

5.4.1.11 #define rad2deg(*angle*)

Convert radians to degrees

5.4.1.12 #define remedy(*a*, *m*)

Return the remedy of a/m division

```
remedy(16,5); // returns 1
remedy(14,5); // returns 4
remedy(14.1,5); // returns 4.1
```

5.4.1.13 #define roundup(*a*, *m*)

Return the positive closest multiple of m less than a

```
roundup(16,5); // returns 15
roundup(14,5); // returns 10
roundup(19,5); // returns 15
```

5.4.1.14 #define square(*x*)

Compute the square of the input parameter

5.4.1.15 #define stringify(*txt*)

Stringify the given parameter or macro expression

5.4.1.16 #define swap(*_type*, *a*, *b*)

Swap the given two variable using the third temporary variable

```
double a = 1.1, b = 2.2;
swap(double, a,b);   // a : 2.2, b : 1.1
```

5.4.1.17 #define va_nargs(*...*)

Counts the number of arguments in variable length argument list. Argument count must be between 1-9.

```
#define fun(...) printf("fun called with %d arguments\n", va_nargs(__VA_ARGS__));
fun(1.2345);  // "fun called with 1 arguments"
fun(1,2);     // "fun called with 2 arguments"
fun(a,b,c,d); // "fun called with 4 arguments"
```

5.5 E:/imlab_library/include/cvcore.h File Reference

File containing example of doxygen usage for quick reference.

```
#include "core.h"
#include "mlcore.h"
```

Data Structures

- struct feature_t
- struct haar_t

Enumerations

- enum cv_algorithm_t {
  CV_ENCODER = 1, CV_LBP = 2, CV_HOG = 3, CV_SIFT = 4,
  CV_NPD = 5 }

Functions

- struct feature_t * feature_create (enum cv_algorithm_t algorithm, uint32_t width, uint32_t height, char *options)

    Construct a feature extraction algorithm with the given parameters.
- uint32_t feature_size (struct feature_t *model)

    Returns the feature size of the model.
- void feature_view (struct feature_t *model)

    Displays the properties of the created model.
- return_t feature_extract (matrix_t *img, struct feature_t *model, float *output)

    Extract the features from the given image and fills the output array.
- return_t feature_image (matrix_t *img, struct feature_t *model, matrix_t *output)

    Visualize the feature if it is possible.
- struct haar_feature_t * haar_feature_create (double thr, int hl, double lv, int ln, int hr, double rv, int rn, int tilt, int length)
- struct haar_tree_t * haar_tree_create (int length)
- struct haar_stage_t * haar_stage_create (double thr, int length)
- struct haar_t * haar_create (int s1, int s2, int length)
- struct haar_t * haar_read (const char *filename)

5.5.1 Detailed Description

File containing example of doxygen usage for quick reference.

Author

My Self

Date

9 Sep 2012 Here typically goes a more extensive explanation of what the header defines. Doxygens tags are words preceeded by either a backslash \ or by an at symbol @.

See Also

http://www.stack.nl/~dimitri/doxygen/docblocks.html
http://www.stack.nl/~dimitri/doxygen/commands.html

5.5.2 Enumeration Type Documentation

5.5.2.1 enum cv_algorithm_t

Enumerator

CV_ENCODER

CV_LBP Local binary patterns are introduced as a texture measure for gray-level images**[ojala1996comparative]**}. LBP is a feature operator that labels each pixel by thresholding with each of their neighbours so that the situation of the all neighbours can be compressed into the single value. The mathematical expression of the LBP operator:

$$LBP_{P,R} = \sum_{p=0}^{P-1} u(g_p - g_c)2^p$$

Here $u(x)$ is a unit function, g_c is the center pixel that LBP operator is applied, g_p is the p^{th} neighbour in R neighbourhood and the P is the number of neighbours.

Generally LBP operator is applied to the given image and then the resulting image is divided into $8 8$ sub-blocks by 4 pixel overlap. For each $8 8$ patch we compute the 64-bin histogram. Concatenating these histogram vectors we construct the feature descriptor.

The following parameters are supported for LBP algorithm

parameter	explanation	default value
`radius`	Radius of the LBP	1
`neighbors`	Number of neighbors to be visited	8
`block`[2]	Block size of the LBP	[8x8]
`uniform`	Integer indicating that uniform LBP (1) or normal LBP (0)	0

An example usage

```
struct feature_t *lbp = feature_create(CV_LBP,
        width(gray),height(gray), "radius:1 neighbors:8 block:[8 8] uniform:0");
feature_view(lbp);
```

CV_HOG Histogram of oriented gradients are commonly used feature descriptors for object detection. HOG is first proposed by Dallal et.al for pedestrian detection **[dalal2005histograms]**} and Singh et.al for gender classification **[singh2013comparison]**}. Similar to the LBP features, HOG keeps weighted orientation of the edges in a small sub-block in a compressed histogram vector.

9-bin HOG for each 8×8 sub-blocks as the feature vector. In order to compress the magnitude and the angle of the orientation in a 9-bin histogram for a given θ angle and M magnitude, we create the weighted histogram vector h for the $bins = [0, 22, 45, 67, 90, ..., 145, 167, 180]$ as follows:

$$b_1 = \arg\min(bins - \theta > 0)$$
$$b_2 = \arg\max(bins - \theta < 0)$$
$$h(b_1) = h(b_1) + M\frac{b_1 - \theta}{b_1 - b_2}$$
$$h(b_2) = h(b_2) + M\frac{\theta - b_2}{b_1 - b_2}$$

After the computation of the histogram for each sub-block, we combine the histograms for every 2×2 sub-block neighbourhood by adding, so we reduce the feature vector size by 4 times.

parameter	explanation	default value
`bins`	Number of bins in the gradient histogram	9
`cell`[2]	Cell size for the histogram of the gradient method. Given image width and height must be divisible bye the cell size	[8x8]
`block`[2]	Block size for the Histogram of the gradient method. Each block is a collection of cells	[2x2]
`stride`[2]	Overlap between the histogram cells	[1x1]

An example usage

```
struct feature_t *hog = feature_create(CV_HOG,
      width(gray),height(gray), "bins:18 block:[6 6] cell:[16 16] stride:[1 1]");
feature_view(hog);
```

CV_SIFT

CV_NPD

5.5.3 Function Documentation

5.5.3.1 struct feature_t* feature_create (enum cv_algorithm_t *algorithm*, uint32_t *width*, uint32_t *height*, char * *options*)

Construct a feature extraction algorithm with the given parameters.

Parameters

algorithm	Algorithm to be used for feature extraction
width	Width of the input image that the features will be extracted
height	Height of the input image that the features will be extracted
options	Optional algorithm parameters. Parameters that are not given by the `options` are used with the default values.

Returns

feature_t object than can be used for feature extraction

5.5.3.2 return_t feature_extract (matrix_t * *img*, struct feature_t * *model*, float * *output*)

Extract the features from the given image and fills the output array.

Parameters

img	Input image that the features will be extracted
model	Input model to be used for feature extraction
output	Output array that the features will be written

Returns

Return success or the corresponding error

5.5.3.3 return_t feature_image (matrix_t * *img*, struct feature_t * *model*, matrix_t * *output*)

Visualize the feature if it is possible.

Parameters

img	Input image that the features will be extracted
model	Input model to be used for feature extraction
output	Output image that the visualized features will be written

Returns

Return success or the corresponding error

5.5.3.4 uint32_t feature_size (struct feature_t * *model*)

Returns the feature size of the model.

Parameters

model	Input model to be used for feature extraction

Returns

Bytes necessary to store the extracted feature

5.5.3.5 void feature_view (struct feature_t * *model*)

Displays the properties of the created model.

Parameters

model	Input model to be displayed

5.5.3.6 struct haar_t* haar_create (int *s1*, int *s2*, int *length*)

5.5.3.7 struct haar_feature_t* haar_feature_create (double *thr*, int *hl*, double *lv*, int *ln*, int *hr*, double *rv*, int *rn*, int *tilt*, int *length*)

5.5.3.8 struct haar_t* haar_read (const char * *filename*)

Reads haar json file and parse the input into the haar_t structure

Parameters

filename	Input file to be read

Returns

Return success or relative error

5.5.3.9 struct haar_stage_t* haar_stage_create (double *thr*, int *length*)

5.5.3.10 struct haar_tree_t* haar_tree_create (int *length*)

5.6 E:/imlab_library/include/imcore.h File Reference

```
#include "core.h"
#include <math.h>
```

Data Structures

- struct color_t

Macros

- #define RGB2GRAY(c) ((uint8_t) ((76 * (uint32_t)(c).red + 151 * (uint32_t)(c).green + 29 * (uint32_t)(c).blue) >> 8))

Functions

- struct color_t RGB (uint8_t r, uint8_t g, uint8_t b)
- struct color_t HSV (uint8_t h, uint8_t s, uint8_t v)
- matrix_t ∗ imread (char ∗filename)
- return_t imload (char ∗filename, matrix_t ∗out)
- return_t imwrite (matrix_t ∗in, char ∗filename)
- return_t matrix2image (matrix_t ∗in, uint8_t opt, matrix_t ∗out)
- struct window_t ∗ window_create (char ∗windowname, uint32_t options)
- void imshow (matrix_t ∗image, struct window_t ∗display)
- int window_wait (struct window_t ∗display, int sleep)
- return_t bwlabel (matrix_t ∗in, matrix_t ∗label, uint32_t ∗numCC)
- return_t imerode (matrix_t ∗in, matrix_t ∗str, matrix_t ∗out)
- return_t imdilate (matrix_t ∗in, matrix_t ∗str, matrix_t ∗out)
- return_t label2rgb (matrix_t ∗in, int inmax, matrix_t ∗out)
- return_t bwdist (matrix_t ∗in, matrix_t ∗out)
- uint32_t imotsu (matrix_t ∗in)
- return_t imbinarize (matrix_t ∗in, uint32_t block_width, uint32_t block_height, float threshold, matrix_t ∗out)
- return_t imthreshold (matrix_t ∗in, uint32_t threshold, matrix_t ∗out)
- return_t rgb2gray (matrix_t ∗in, matrix_t ∗out)
- return_t gray2rgb (matrix_t ∗in, matrix_t ∗out)
- return_t rgb2hsv (matrix_t ∗in, matrix_t ∗out)
- return_t hsv2rgb (matrix_t ∗in, matrix_t ∗out)
- return_t rgb2any (matrix_t ∗in, const float tr[3][4], matrix_t ∗out)
- void rgb2lab (struct color_t C1, float ∗L, float ∗a, float ∗b)
- return_t image_set (matrix_t ∗in, uint32_t y, uint32_t x, struct color_t clr)
- return_t image_get (matrix_t ∗in, uint32_t y, uint32_t x, struct color_t ∗clr)
- return_t imcrop (matrix_t ∗in, struct rectangle_t crop_region, matrix_t ∗out)
- return_t imresize (matrix_t ∗in, uint32_t nwidth, uint32_t nheight, matrix_t ∗out)
- return_t imscale2x (matrix_t ∗in, matrix_t ∗out)
- matrix_t ∗ maketform (float data[9])
- matrix_t ∗ rot2tform (float cx, float cy, float theta, float scale)
- matrix_t ∗ pts2tform (struct point_t ∗src, struct point_t ∗dst, int len)
- return_t imtransform (matrix_t ∗in, matrix_t ∗tform, matrix_t ∗out)
- return_t integral (matrix_t ∗in, matrix_t ∗sums, matrix_t ∗ssum)
- return_t medfilt2 (matrix_t ∗input, uint32_t filter_width, uint32_t filter_height, matrix_t ∗output)
- float integral_get_float (matrix_t ∗in, int x0, int y0, int x1, int y1)
- double integral_get_double (matrix_t ∗in, int x0, int y0, int x1, int y1)
- void draw_point (matrix_t ∗img, struct point_t p1, struct color_t clr, int thickness)
- void draw_line (matrix_t ∗img, struct point_t p1, struct point_t p2, struct color_t clr, int thickness)
- void draw_rectangle (matrix_t ∗img, struct rectangle_t r, struct color_t clr, int thickness)

5.6.1 Macro Definition Documentation

5.6.1.1 #define RGB2GRAY(*c*) ((uint8_t) ((76 ∗ (uint32_t)(c).red + 151 ∗ (uint32_t)(c).green + 29 ∗ (uint32_t)(c).blue) >> 8))

5.6.2 Function Documentation

5.6.2.1 return_t bwdist (matrix_t ∗ *in,* matrix_t ∗ *out*)

5.6.2.2 return_t bwlabel (matrix_t ∗ *in,* matrix_t ∗ *label,* uint32_t ∗ *numCC*)

5.6.2.3 void draw_line (matrix_t ∗ *img,* struct point_t *p1,* struct point_t *p2,* struct color_t *clr,* int *thickness*)

5.6.2.4 void draw_point (**matrix_t** ∗ *img,* struct **point_t** *p1,* struct **color_t** *clr,* int *thickness*)

5.6.2.5 void draw_rectangle (**matrix_t** ∗ *img,* struct **rectangle_t** *r,* struct **color_t** *clr,* int *thickness*)

5.6.2.6 **return_t** gray2rgb (**matrix_t** ∗ *in,* **matrix_t** ∗ *out*)

5.6.2.7 struct **color_t** HSV (uint8_t *h,* uint8_t *s,* uint8_t *v*)

5.6.2.8 **return_t** hsv2rgb (**matrix_t** ∗ *in,* **matrix_t** ∗ *out*)

5.6.2.9 **return_t** image_get (**matrix_t** ∗ *in,* uint32_t *y,* uint32_t *x,* struct **color_t** ∗ *clr*)

5.6.2.10 **return_t** image_set (**matrix_t** ∗ *in,* uint32_t *y,* uint32_t *x,* struct **color_t** *clr*)

5.6.2.11 **return_t** imbinarize (**matrix_t** ∗ *in,* uint32_t *block_width,* uint32_t *block_height,* float *threshold,* **matrix_t** ∗ *out*)

5.6.2.12 **return_t** imcrop (**matrix_t** ∗ *in,* struct **rectangle_t** *crop_region,* **matrix_t** ∗ *out*)

5.6.2.13 **return_t** imdilate (**matrix_t** ∗ *in,* **matrix_t** ∗ *str,* **matrix_t** ∗ *out*)

5.6.2.14 **return_t** imerode (**matrix_t** ∗ *in,* **matrix_t** ∗ *str,* **matrix_t** ∗ *out*)

5.6.2.15 **return_t** imload (char ∗ *filename,* **matrix_t** ∗ *out*)

Load the given image file into the given output matrix. The output matrix should have the same size with the given image. This method is useful if you read images with the same & known sizes in a loop.

Parameters

filename	Input image to be read
out	Output matrix to be filled

Returns

5.6.2.16 uint32_t imotsu (**matrix_t** ∗ *in*)

5.6.2.17 **matrix_t**∗ imread (char ∗ *filename*)

Read the given image file and return the output matrix.

Parameters

filename	Input image to be read

Returns

5.6.2.18 **return_t** imresize (**matrix_t** ∗ *in,* uint32_t *nwidth,* uint32_t *nheight,* **matrix_t** ∗ *out*)

5.6.2.19 **return_t** imscale2x (**matrix_t** ∗ *in,* **matrix_t** ∗ *out*)

5.6.2.20 void imshow (**matrix_t** ∗ *image,* struct window_t ∗ *display*)

Display the given image onto the given window

Parameters

image	IMLAB image to be displayed
display	Window object which is allocated by window_create

5.6.2.21 return_t imthreshold (matrix_t ∗ *in*, uint32_t *threshold*, matrix_t ∗ *out*)

5.6.2.22 return_t imtransform (matrix_t ∗ *in*, matrix_t ∗ *tform*, matrix_t ∗ *out*)

5.6.2.23 return_t imwrite (matrix_t ∗ *in*, char ∗ *filename*)

Write the image into the given filename

Parameters

in	Input image to be written
filename	Output filename for the image

Returns

5.6.2.24 return_t integral (matrix_t ∗ *in*, matrix_t ∗ *sums*, matrix_t ∗ *ssum*)

5.6.2.25 double integral_get_double (matrix_t ∗ *in*, int *x0*, int *y0*, int *x1*, int *y1*)

5.6.2.26 float integral_get_float (matrix_t ∗ *in*, int *x0*, int *y0*, int *x1*, int *y1*)

5.6.2.27 return_t label2rgb (matrix_t ∗ *in*, int *inmax*, matrix_t ∗ *out*)

5.6.2.28 matrix_t∗ maketform (float *data[9]*)

5.6.2.29 return_t matrix2image (matrix_t ∗ *in*, uint8_t *opt*, matrix_t ∗ *out*)

5.6.2.30 return_t medfilt2 (matrix_t ∗ *input*, uint32_t *filter_width*, uint32_t *filter_height*, matrix_t ∗ *output*)

5.6.2.31 matrix_t∗ pts2tform (struct point_t ∗ *src*, struct point_t ∗ *dst*, int *len*)

5.6.2.32 struct color_t RGB (uint8_t *r*, uint8_t *g*, uint8_t *b*)

5.6.2.33 return_t rgb2any (matrix_t ∗ *in*, const float *tr[3][4]*, matrix_t ∗ *out*)

5.6.2.34 return_t rgb2gray (matrix_t ∗ *in*, matrix_t ∗ *out*)

5.6.2.35 return_t rgb2hsv (matrix_t ∗ *in*, matrix_t ∗ *out*)

5.6.2.36 void rgb2lab (struct color_t *C1*, float ∗ *L*, float ∗ *a*, float ∗ *b*)

5.6.2.37 matrix_t∗ rot2tform (float *cx*, float *cy*, float *theta*, float *scale*)

5.6.2.38 struct window_t∗ window_create (char ∗ *windowname*, uint32_t *options*)

Create a window to display the image

Parameters

windowname	Window name to be displayed on the window name bar
options	Window options (Not supported).

Returns

Return a window object that can imshow can write images onto it

5.6.2.39 int window_wait (struct window_t ∗ *display,* int *sleep*)

Wait for #sleep milliseconds to display the image.

Parameters

display	Window object which is allocated by window_create
sleep	Sleep duration to show the image

Returns

Return the pressed key or mouse button during the wait

5.7 E:/imlab_library/include/iocore.h File Reference

```
#include "core.h"
```

Macros

IMLAB JSON get mehods

These are the definitions of the public functions for getting each element from the object or array. Use these functions if you need any scalar or array type element from any json array or object. These functions are overloaded based on the number of input arguments. So use json_get_X(data, name, id) to get a data from the object or json_get_X(data, id) from the array.

- #define json_get(...)
  This function will return the json_t∗ holded in object->element["name"].
- #define json_get_boolean(...)
  This function will return the boolean (int8_t) holded in the object->element["name"].
- #define json_get_number(...)
  This function will return the string (char∗) holded in the object->element["name"].
- #define json_get_string(...)
  This function will return the number (double) holded in the object->element["name"].
- #define json_get_array(...)
  This function will return the array (struct json_array_t∗) holded in the object->element["name"].
- #define json_get_object(...)
  This function will return the object (struct json_object_t∗) holded in the object->element["name"].

Functions

- struct csv_t ∗ csv_open (char ∗filename, uint32_t skip_rows, uint32_t buffer_size)
- uint32_t csv_get_column_size (struct csv_t ∗in)
- int csv_get_next_line (struct csv_t ∗in)
- long csv_get_long (struct csv_t ∗in, uint32_t col)
- char ∗ csv_get_string (struct csv_t ∗in, uint32_t col)

- double csv_get_double (struct csv_t ∗in, uint32_t col)
- void csv_close (struct csv_t ∗∗out)
- return_t imlab_mkdir (const char ∗pathname)
- char ∗ imlab_filename (const char ∗filename, const char ∗extension)
- vector_t ∗ json_read (const char ∗filename)
- return_t json_write (const char ∗filename, vector_t ∗input)
- return_t json_serialize (vector_t ∗input, string_t ∗buffer)
- return_t json_free (vector_t ∗∗in)
- vector_t ∗ json_create (void)

5.7.1 Detailed Description

Here the defintions of the Input/output functions provided by the IMLAB library.

5.7.2 Macro Definition Documentation

5.7.2.1 #define json_get(...)

This function will return the json_t∗ holded in object->element["name"].

5.7.2.2 #define json_get_array(...)

This function will return the array (struct json_array_t∗) holded in the object->element["name"].

5.7.2.3 #define json_get_boolean(...)

This function will return the boolean (int8_t) holded in the object->element["name"].

5.7.2.4 #define json_get_number(...)

This function will return the string (char∗) holded in the object->element["name"].

5.7.2.5 #define json_get_object(...)

This function will return the object (struct json_object_t∗) holded in the object->element["name"].

5.7.2.6 #define json_get_string(...)

This function will return the number (double) holded in the object->element["name"].

5.7.3 Function Documentation

5.7.3.1 void csv_close (struct csv_t ∗∗ *out*)

Close the given file and deallocates the memory

Parameters

out	Address of the pointer to the csv structure

5.7.3.2 uint32_t csv_get_column_size (struct csv_t * *in*)

Returns the number of columns in the current csv file

Parameters

in	Input pointer to the csv structure

Returns

Number of columns in the file

5.7.3.3 double csv_get_double (struct csv_t ∗ *in*, uint32_t *col*)

Get the double value in the current row and given column

Parameters

in	Input pointer to the csv structure
col	Column index of the double data (0 based)

Returns

Double value of the cell (current row, col)

5.7.3.4 long csv_get_long (struct csv_t ∗ *in*, uint32_t *col*)

Get the integer value in the current row and given column

Parameters

in	Input pointer to the csv structure
col	Column index of the integer data (0 based)

Returns

Integer value of the cell (current row, col)

5.7.3.5 int csv_get_next_line (struct csv_t ∗ *in*)

Scan the next row of the current file and parse the cells

Parameters

in	Input pointer to the csv structure

Returns

Return the number scanned line (zero or one)

5.7.3.6 char∗ csv_get_string (struct csv_t ∗ *in*, uint32_t *col*)

Get the string value in the current row and given column

Parameters

in	Input pointer to the csv structure
col	Column index of the string data (0 based)

Returns

String value of the cell (current row, col)

5.7.3.7 struct csv_t∗ csv_open (char ∗ *filename,* uint32_t *skip_rows,* uint32_t *buffer_size*)

CSV is simple but very efficient CSV (Comma Seperated Values) reader library designed for IMLAB Image Processing Library. It has has been written in C89 standart and in the same exact way with the IMLAB library. It is designed to be readable, easy to use and easy to remember. It has a single header (import and use) file and all the functions are created under the json_ namespace. A simple code written with CSV looks like this.

```
struct csv_t *table = csv_open("sample.csv", 0, 1024);
int cols = csv_get_column_size(table);

// scan the header line
csv_get_next_line(table);

// print the header
for(int i = 0; i < cols; i++)
 {
    printf("%s \t ", csv_get_string(table, i));
 }
printf("\n");
// print the values
while(csv_get_next_line(table))
 {
    int id = csv_get_long(table, 0);
    char *name = csv_get_string(table, 1);
    double score = csv_get_double(table, 2);

    printf("[%02d] \t [%s] \t [%3.2f]\n", id, name, score);
 }

csv_close(&table);
```

Open a csv file and return the handler

Parameters

filename	Input filename with csv extension
skip_rows	Number of rows to be skipped
buffer_size	Max length of the single cell element in bytes (typical: 1024)

Returns

A pointer to the created csv structure

5.7.3.8 char∗ imlab_filename (const char ∗ *filename,* const char ∗ *extension*)

Create and return a unique filename that is not in the current directory.

Parameters

filename	Name of the file
extension	Extension of the filename

Returns

filename_d.extension formatted char pointer

5.7.3.9 return_t imlab_mkdir (const char ∗ *pathname*)

Create a directory with the given name and set the mode to the given mode

Parameters

pathname	Name of the path

Returns

5.7.3.10 vector_t∗ json_create (void)

This is a constructor for JSON null object. This is usefull in order to prevent errors.

5.7.3.11 return_t json_free (vector_t ∗∗ *in*)

Free up the memory holded by the json object

Parameters

in	Input json object to be freed

Returns

SUCCESS if the cleaning is succesfull

5.7.3.12 vector_t∗ json_read (const char ∗ *filename*)

JSON is simple but very efficient JSON reader library designed for IMLAB Image Processing Library. It has has been written in C89 standart and in the same exact way with the IMLAB library. It is designed to be readable, easy to use and easy to remember. It has a single header (import and use) file and all the functions are created under the json_ namespace. A simple code written with JSON looks like this.

```
vector_t *root = json_read("haarcascade_frontalface_alt.json");
int s = length(root);

int s1 = json_get_number(root, "size1", 0);
int s1 = json_get_number(root, "size2", 0);
printf("Cascade Box Size: %d x %d\n", s1, s2);

// we can get stages array from the current node using json_get_array
vector_t *stages = json_get_array(root, "stages", 0);

// stages is an array which holds a number and an array inside
for(i=0; i < length(stages); i++) {
    // while getting item from array just use two variable array name and index
    vector_t *obj = json_get_object(stages, i);
    double stage_threshold = json_get_number(obj, "thres", 0);
    vector_t *trees = json_get_array(obj, "trees", 0);
    for(j=0; j < length(trees); j++) {
        ...
    }
}
```

As seen in the sample code, any JSON object or array is holded by the vector_t structure. In order to hold different types in a single vector, IMLAB defines a new structure namely struct json_data_t. This structure is consist of a type identifier and a void pointer to the real data. A JSON data could be created with the following constructors:

```
vector_t *root = vector_create(struct json_data_t, 10);

struct json_data_t temp;

// sets the temp to JSON boolean 0 (false)
json_boolean(0, &temp);
// inserts it into the root array
vector_push(root, &temp);
```

```
// sets the temp to 1.27
json_number(1.27, &temp);
// inserts it into the root array
vector_push(root, &temp);

// sets the temp to JSON string
json_string("This is a sample JSON string!", &temp);
// inserts it into the root array
vector_push(root, &temp);

// sets the temp to JSON object
vector_t *object = vector_create(struct json_object_t, 1);
json_object(object, &temp);
// inserts it into the root array
vector_push(root, &temp);

// sets the temp to JSON object
vector_t *array = vector_create(struct json_data_t, 1);
json_array(array, &temp);
// inserts it into the root array
vector_push(root, &temp);
```

An array element can hold number of different typed objects under its element pointer. A JSON array can be constructed by vector_create(struct json_data_t). struct json_data_t is the key type for the all JSON library. It has a type indicator and a generic value holder. This propoerty of json_data_t provides the JSON array to hold any data type.

JSON array is consist of pointer to the generic json data holder type json_t∗. Each data can be accessed via its position in the array. IMLAB JSON library provides the following helpful macros to get or set any value inside the array.

- json_push(vector_t∗, data)
- json_get(vector_t∗, id)
- json_get_boolean(vector_t∗, id)
- json_get_string(vector_t∗, id)
- json_get_number(vector_t∗, id)
- json_get_array(vector_t∗, id)
- json_get_object(vector_t∗, id) An object element can hold a number of different type object and their names under its element pointer. A JSON object can be constructed by json_object_create().
- json_object_create(int capacity) : This function creates an json_object_t object which can store capacity json_t∗ inside

JSON object is consist of pointer to the generic json data holder type json_t∗. Each data can be accessed by its name or id. IMLAB JSON library provides the following helpful macros to get or set any value inside the object.

- json_push(json_t∗, data, name)
- json_get(json_t∗, name, id)
- json_get_id(object, name)
- json_get_name(object, id)
- json_get_boolean(json_t∗, name, id)
- json_get_string(json_t∗, name, id)
- json_get_number(json_t∗, name, id)
- json_get_array(json_t∗, name, id)
- json_get_object(json_t∗, name, id)

If the object name is empty than the function will use the given id as the index and return the value. This could be faster if the object has too many data elements and you already know the index of the element. This data structure will hold a single data element which should have a type name and value. JSON_T is the most generic container for any json type object. It holds a single data element which should have a type name and value. Value is defined as void pointer so that it can hold all types of data without any problem. In the user side this type is the generic input type for all the json functions and return types for most the json functions (excepts for the type specified functions).

The following functions accepts the json_t∗ input:

- json_get_length(json_t∗) : This function return the length of the given data. If the data is static (number, string, primitive) it returns 1 otherwise the length of the given data.
- json_get_[boolean, number, string, array, object](json_t∗, name, id) : Returns the data holded in the elements of the given data (object) with the specified type.
- json_get_[boolean, number, string, array, object](json_t∗, id) : Returns the data holded in the elements of the given data (array) with the specified type. This is the reader function for any json file. This function reads the json data in the given file and parse the all name value pairs into the json_t∗ data.

5.7.3.13 return_t json_serialize (vector_t ∗ *input,* string_t ∗ *buffer*)

Serialize the given JSON structure into the string buffer

Parameters

input	JSON structure created by json_create or json_read.
buffer	Empty string pointer which will be filled by the function.
buffer_length	Initial buffer length which will be updated by the function.

Returns

Return positive if no error occurs.

5.7.3.14 return_t json_write (const char ∗ *filename,* vector_t ∗ *input*)

This is the serializer function for any json file. This function reads the parsed json data and serialize it into the given file.

5.8 E:/imlab_library/include/lacore.h File Reference

```
#include <stdint.h>
#include "core.h"
```

Functions

- float fast_atan2f (float x, float y)

5.8.1 Function Documentation

5.8.1.1 float fast_atan2f (float *x,* float *y*) [inline]

5.9 E:/imlab_library/include/mlcore.h File Reference

```
#include <omp.h>
#include <math.h>
#include "core.h"
#include "prcore.h"
```

Macros

- #define IM_MAX_MULTI_CLASS 100
- #define LSRL1 0
- #define LSRL2 1
- #define SVRL1 2
- #define SVRL2 3
- #define IM_CLASSORREG 4.5
- #define LREL1 6
- #define LREL2 7
- #define SVML1 8
- #define SVML2 9
- #define FERN_CUSTOM_REGRESSOR 0
- #define FERN_CUSTOM_CLASSIFIER 1
- #define FERN_RANDOM_REGRESSOR 2
- #define FERN_CORRELATION_REGRESSOR 3
- #define FERN_RANDOM_CLASSIFIER 4
- #define FERN_CORRELATION_CLASSIFIER 5

Functions

- struct glm_t ∗ glm_create (uint32_t solver, char options[ ])

    glm_t Functions are included
- return_t glm_view (struct glm_t ∗model)
- return_t glm_train (matrix_t ∗in, vector_t ∗label, struct glm_t ∗model)

    HIGH LEVEL FUNCTIONS TO CALL FROM THE IMLAB APPLICATIONS.
- return_t glm_predict (matrix_t ∗in, matrix_t ∗label, struct glm_t ∗model)
- struct glm_t ∗ glm_read (const char ∗filename)

    glm_t model import/export functions
- return_t glm_write (struct glm_t ∗net, const char ∗filename)
- struct fern_t ∗ fern_create (uint8_t solver, char options[ ])
- return_t fern_set (struct fern_t ∗model, return_t(∗select)(float ∗, float ∗, float ∗, void ∗∗, uint32_t, uint32_t, uint32_t, uint32_t, uint32_t ∗∗, float ∗∗, uint32_t), uint32_t(∗binarize)(float ∗data, uint32_t ∗p, float ∗t), void ∗aux)
- return_t fern_train (matrix_t ∗input, matrix_t ∗output, struct fern_t ∗model)

    HIGH LEVEL FUNCTIONS TO CALL FROM THE IMLAB APPLICATIONS.
- return_t fern_predict (matrix_t ∗in, matrix_t ∗output, struct fern_t ∗model)
- matrix_t ∗ fern_visualize (uint32_t out_width, uint32_t out_height, uint32_t scale, struct fern_t ∗model)
- struct fern_t ∗ fern_read (const char ∗filename)
- return_t fern_write (struct fern_t ∗model, const char ∗filename)

5.9.1 Macro Definition Documentation

5.9.1.1 #define FERN_CORRELATION_CLASSIFIER 5

5.9.1.2 #define FERN_CORRELATION_REGRESSOR 3

5.9.1.3 #define FERN_CUSTOM_CLASSIFIER 1

5.9.1.4 #define FERN_CUSTOM_REGRESSOR 0

5.9.1.5 #define FERN_RANDOM_CLASSIFIER 4

5.9.1.6 #define FERN_RANDOM_REGRESSOR 2

5.9.1.7 #define IM_CLASSORREG 4.5

5.9.1.8 #define IM_MAX_MULTI_CLASS 100

5.9.1.9 #define LREL1 6

5.9.1.10 #define LREL2 7

5.9.1.11 #define LSRL1 0

5.9.1.12 #define LSRL2 1

5.9.1.13 #define SVML1 8

5.9.1.14 #define SVML2 9

5.9.1.15 #define SVRL1 2

5.9.1.16 #define SVRL2 3

5.9.2 Function Documentation

5.9.2.1 struct fern_t∗ fern_create (uint8_t *solver,* char *options[ ]*)

5.9.2.2 return_t fern_predict (matrix_t ∗ *in,* matrix_t ∗ *output,* struct fern_t ∗ *model*)

5.9.2.3 struct fern_t∗ fern_read (const char ∗ *filename*)

5.9.2.4 return_t fern_set (struct fern_t ∗ *model,* return_t(∗)(float ∗, float ∗, float ∗, void ∗∗, uint32_t, uint32_t, uint32_t, uint32_t, uint32_t ∗∗, float ∗∗, uint32_t) *select,* uint32_t(∗)(float ∗data, uint32_t ∗p, float ∗t) *binarize,* void ∗ *aux*)

5.9.2.5 return_t fern_train (matrix_t ∗ *input,* matrix_t ∗ *output,* struct fern_t ∗ *model*)

HIGH LEVEL FUNCTIONS TO CALL FROM THE IMLAB APPLICATIONS.

5.9.2.6 matrix_t∗ fern_visualize (uint32_t *out_width,* uint32_t *out_height,* uint32_t *scale,* struct fern_t ∗ *model*)

5.9.2.7 return_t fern_write (struct fern_t ∗ *model,* const char ∗ *filename*)

5.9.2.8 struct glm_t∗ glm_create (uint32_t *solver,* char *options[ ]*)

glm_t Functions are included

5.9.2.9 return_t glm_predict (matrix_t ∗ *in*, matrix_t ∗ *label*, struct glm_t ∗ *model*)

5.9.2.10 struct glm_t∗ glm_read (const char ∗ *filename*)

glm_t model import/export functions

5.9.2.11 return_t glm_train (matrix_t ∗ *in*, vector_t ∗ *label*, struct glm_t ∗ *model*)

HIGH LEVEL FUNCTIONS TO CALL FROM THE IMLAB APPLICATIONS.

5.9.2.12 return_t glm_view (struct glm_t ∗ *model*)

5.9.2.13 return_t glm_write (struct glm_t ∗ *net*, const char ∗ *filename*)

5.10 E:/imlab_library/include/prcore.h File Reference

```
#include <omp.h>
#include <math.h>
#include "core.h"
```

Macros

- #define random_max 4294967295

Functions

- int32_t ∗ range (int32_t start, int32_t end)
- float pearson (const float ∗vecA, const float ∗vecB, int length)
- float covariance (const float ∗vecA, const float ∗vecB, int length)
- void random_seed ()
- void random_setseed (uint32_t state[4])
- void random_getseed (uint32_t state[4])
- uint32_t random ()
- int random_int (int mint, int maxt)
- float random_float (float mint, float maxt)
- uint32_t ∗ random_permutation (uint32_t length)
- void ∗ random_sample (uint8_t ∗in, uint32_t length, uint32_t sample, uint32_t elemsize)

5.10.1 Macro Definition Documentation

5.10.1.1 #define random_max 4294967295

Maximum number can be genarted by IMLAB Random class_name

5.10.2 Function Documentation

5.10.2.1 float covariance (const float ∗ *vecA*, const float ∗ *vecB*, int *length*)

5.10.2.2 float pearson (const float ∗ *vecA*, const float ∗ *vecB*, int *length*)

5.10.2.3 uint32_t random ( )

Wikipedia implementation of xorshift128 algorithm. This will produce a random integer in $[0, 2^32 - 1]$ range with a period of $2^128 - 1$. Which is quite unpredictable and random for most of the applications.

5.10.2.4 float random_float (float *mint,* float *maxt*)

5.10.2.5 void random_getseed (uint32_t *state[4]*)

Get the current seed state to replicate PRNG process later.

5.10.2.6 int random_int (int *mint,* int *maxt*)

Generate a random integer in [mint, maxt] range with a uniform distribution.

5.10.2.7 uint32_t∗ random_permutation (uint32_t *length*)

5.10.2.8 void∗ random_sample (uint8_t ∗ *in,* uint32_t *length,* uint32_t *sample,* uint32_t *elemsize*)

5.10.2.9 void random_seed ( )

Create a random seed for the random number generator.

5.10.2.10 void random_setseed (uint32_t *state[4]*)

Set the seed for the random number generator. This is useful when you need to generate controlled random numbers. Avoid using 0 as the initial seed.

5.10.2.11 int32_t∗ range (int32_t *start,* int32_t *end*)

Initial seed for the random number generator.

Index