# Getting to Know OpenCV Data Types

## ★ The Basics (OpenCV Data Types)

⇒ OpenCV has many data types, which are designed to make the representation and handling of important computer vision concepts relatively easy and intuitive.

## ★ Overview of the Basic Types

cv::Vec<> → fixed vector classes (handle small vector)

⇒ Aliases of cv::Vec<> :

- cv::Vec2i → two element integer vector
- cv::Vec3i → three element integer vector
- cv::Vec4d → four-element double-precision floating-point vector

⇒ In general, anything of the form cv::Vec{2,3,4,6}{b,w,s,i,f,d} is a valid combination.

⇒ In addition to the fixed vector classes, there are also fixed matrix classes.

cv::Matx<> → {Designed for small Matrix}

⇒ Aliases of cv::Matx<> :

cv::Matx{1,2,3,4,6}{1,2,3,4,6}{f,d}

⇒ The dimensionality of the fixed matrix classes must be known at compile time.

Point classes

↳ The main difference between the point class and fixed vector class is that their members are accessed by named variables (mypoint.x, mypoint.y etc) rather than by a vector index (myvec[0], myvec[1] etc..).

⇒ Aliases are :-

cv::Point2i    cv::Point2f    cv::Point2d

cv::Point3i    cv::Point3f    cv::Point3d

⇒ The class cv::Scalar is essentially a four-dimensional point.

↳ aliased to a four component vector with double precision.

cv::Vec<double,4>

⇒ cv::Size

↳ It has data member width and height.

cv::Size2i    cv::Size2f

cv::Rect

↳ has all four.

## * Basic Type: Getting down to details

### # The point class

b → Unsigned character
S → Short integer
i → Integer
f → 32-bit floating-point number
d → 64-bit floating-point number

⇒ Important operations.

P.x , P.y , P.z {Member access}

x = P1.dot(P2) {Dot product}

x = P1.ddot(P2) {Double-precision dot-product}

P1.cross(P2) {Cross product (only for 3D point)}

P.inside(r) {Query if point P in inside rectangle r, Only for 2D point}

### # The cv::Scalar class

⇒ The cv::Scalar class also has some special member functions associated with uses of four-component vectors in computer vision.

⇒ Important operations.

s1.mul(s2) {Element-wise multiplication}

s.conj() {Quaternion Conjugation}

s.isReal() {Quaternion real test}

⇒ cv::Scalar inherits directly from an instantiation of the fixed vector class template.

↳ As a result, it inherits all of the vector ~~class~~ algebra operations.

# The size classes

⇒ Important operations:

sz.area(); {Computes area}

sz.width sz.height {Member access}

# The cv::Rect class

⇒ The rectangle classes include the member x and y of the point class (representing the upper-left corner of the rectangle) and the members width and height of the size class (representing the rectangle's size).

r.x ; r.y ; r.width ; r.height {Member access}

r.area() {Compute area}

r.tl() {Extract upper-left corner}

r.br() {Extract bottom right corner}

r.contains(p) {Determine if point p is inside rectangle or}

⇒ Many overloaded operators are available for cv::Rect.

# The cv::RotatedRect class

⇒ It is a Container that holds

↳ A cv::Point2f called center

↳ A cv::Size2f called size

↳ and one additional float called angle.

{rotation of rect around center}

rr.center ; rr.size ; rr.angle {Member access}

rr.points (pts[4]) {Returns a list of the corners}

# The fixed matrix classes

⇒ In general, you should use the fixed matrix classes when you are representing something that is really a matrix with which you are going to do matrix algebra.

⇒ Important operations:

cv::Matx21f m(x0, x1) {Value Constructor}

m32f = cv::Matx32f::all(x) {Matrix of all identical elements}

m32f = cv::Matx32f::zeros() {Matrix of zeros}

m32f = cv::Mat32f::ones() {Matrix of all ones}

Scanned by CamScanner

m33f = cv::Matx33f::eye(); {Create a unit matrix}

m33f = cv::Matx33f::randu (min, max)

{Create a matrix with uniformly distributed entries}

m33f = cv::Matx33f::nrandn (mean, variance)

{Create a matrix with normally distributed entries.}

m(i,j) m(i) {Member access}

m1 = m0; m0 * m1
m0+m1 ; m0 - m1 {Matrix algebra}

m*a ; a * m ; m/a {Singleton algebra}

m1 == m2 ; m1 != m2 {Comparision}

m1. dot (m2); {Dot product precision of m}

m1. ddot (m2): {Dot product double precision}

m91f = m33f. reshape<9,1>(); {Reshape a matrix}

m44f = (Matx44f) m44d; {Cast operators}

m44f. get_minor<2,2> (i,j); {Extract 2x2 submatrix at (i,j)}

m14 = m44f. row(i); {Extract row i}

m41f = m44f. col(j); {Extract Column j}

m41f = m44f. diag(); {Extract matrix diagonal}

m44f = m44f.t()  {Compute transpose}

m44f = m44f.inv(method)

{Compute inverse}  {Default → cv::DECOMP_LU}

m31f = m33f.Solve(rhs31f, method)
m32f = m33f.Solve<2>(rhs32f, method)

{Solve linear system}

m1.mul(m2);  {Per element multiplication}

# The fixed vector class

⇒ The vector class are derived from the fixed matrix class.

cv::Vec{2,3,4,6}{b,s,w,i,f,d}

⇒ Important functions:

v4f[i], v3w(j)  {Member access}

v3f.cross(u3f)  {Vector cross product}

# The Complex number class

⇒ Important operations:

cv:: Complexd(re0, im1); {Value Constructor}

~~cv:: Complexf~~

Z1.re   Z1.im   {Member access}

Z2 = Z1.conj()   {Complex conjugate}

# ★ Helper Objects

## # The cv::TermCriteria class

⇒ They have three public member ~~function~~ variables

- type (int)
- max count (int)
- epsilon (double)

⇒ Variables can be set directly or using the following constructor.

TermCriteria (int type, int maxCount, double epsilon)

type can be
- cv::TermCriteria:: COUNT
- cv:: TermCriteria:: EPS
- you can also use them together (i.e. |).

Scanned by CamScanner

# The cv::Range class

⇒ The cv::Range class is used to specify a continuous sequence of integers.

⇒ cv::Range object has two elements, start and end.

cv::Range (int start, int end) {Constructor}

⇒ Ranges are inclusive of their start value, but not inclusive of their end value.

⇒ using size(), you can find the number of elements in a range.

→ member function empty() to check if range has no element.

# The cv::Ptr template and Garbage Collection 101

⇒ First you define an instance of the pointer template for the class object that you want to "wrap".

Example

cv::Ptr <Matx33f> p (new cv::Matx33f)

or

cv::Ptr <Matx33f> p = makePtr<cv::Matx33f>();

{The Constructor for the template object takes a pointer to the object to be pointed to.}

⇒ Modern version of loading image:

```
cv::Ptr<IplImage> img_p (cvLoadImage("an_Image"));
```

\# The cv:: Exception class and exception handling

⇒ opencv use exceptions to handle errors.

⇒ OpenCV defines its own exception type, cv::Exception, which is derived from the STL exception class std::exception.

⇒ The type cv::Exception has members:

① code ⇒ A numerical error code.

② err ⇒ A string indicating the nature of the error that generated the exception.

③ func ⇒ The name of the function in which error occurred.

④ file ⇒ The file in which the error occurred.

⑤ line ⇒ An integer indicating line on which the error occurred in that file.

⇒ There are several built-in macros for generating exceptions yourself.

CV_Error(errorcode, description)

→ Generate and throw an exception with a fixed text description

CV_Error_(errorcode, printf_fmt_str, [printf-args])

Works the same, but allows you to replace the fixed description with a printf-like format string and arguments.

CV_Assert (condition)

Throw an exception if the condition is not met.

CV_DbgAssert (condition)

Same as above but will only operate in debug build.

⇒ These macros are the strongly preferred method of throwing exceptions as they will automatically take care of the fields func, file and line for you.

# The cv::DataType<> template

TODO

# The cv::InputArray and cv::OutputArray

⇒ The primary difference between cv::InputArray and cv::OutputArray is that the former is assumed to be const (i.e. read only)

# Utility Functions

⇒ The utility functions include tools for mathematical operations, tests, error generations, memory and thread handling, optimization, and more.

# cv::alignPtr()

template<T> T* cv::alignPtr(T* ptr, int n = sizeof(T))

- cv::alignSize()
- cv::allocate() → Similar to array form of new.
- cv::deallocate() → Similar to array form of delete.

# cv::fastAtan2()

⇒ This function computes the arctangent of an x,y pair and returns the angle from the origin to the indicated point.

⇒ Results reported in degree ranging from 0.0 to 360.0, inclusive of 0.0 but not inclusive of 360.0.

# cvCeil()

# cv::cubeRoot()

# cv::CV_Assert and CV_DbgAssert() → {Both are macros}

CV_Assert → {If expression evaluates to False (or 0), it will throw an exception.} → Always tested

CV_DbgAssert → {Same but only tested in debug compilations}

# cv::CV_Error() and CV_Error_() → {Both are macro}

# cv::fastFree()

⇒ deallocates buffers that were allocated with cv::fastMalloc().

# cv::fastMalloc()

→ {Works just like the malloc() you are familiar with, except that it is often faster, and it does buffer size alignment for you.}

# cvFloor()

# cv::format()

→ {Similar to sprintf() from the standard library}

# cv::getCPUTickCount()

⇒ This function is best for tasks like initializing random number generators.

⇒ Use cv::getTickCount() for timing measurements.

# cv::getNumThreads()

⇒ Returns the current number of threads used by OpenCV.

# cv::getTickCount

⇒ This function returns a tick count relative to some architecture-dependent time.

⇒ The rate of ticks is also architecture and operating system dependent, however; the time per tick can be computed by cv::getTickFrequency()

# cv:: getTickFrequency()

# cvIsInf()

⇒ Returns 1 if $x$ is $\pm\infty$ and 0 otherwise.

⇒ According to IEEE754 standard.

# cvIsNaN()

⇒ Returns 1 if $x$ is not a number and 0 otherwise.

⇒ According to IEEE 754 standard.

# cvRound()

⇒ Returns closest integer.

# cv:: setNumThreads()

# cv:: setUseOptimized()

# cv:: useOptimized()

Scanned by CamScanner