4

Images and Large Array Types

★ Dynamic and Variable Storage

⇒ `cv::Mat` is considered the epicenter of the entire C++ implementation of the OpenCV library.

⇒ The overwhelming majority of functions in the OpenCV library are :-
- Members of the `cv::Mat` class
- takes `cv::Mat` as an argument
- Or returns `cv::Mat` as a return value

⇒ The `cv::Mat` class is used to represents dense arrays of any number of dimensions.
  - The alternative would be a sparse array.
    - On the case of a sparse array, only non-zero entries are typically stored.
      - This can result in a great savings of storage space if many of the entries are infact zero, but can be very wasteful if the array is relatively dense.
        - Common use case of sparse array rather dense array would be histograms.
          - `cv::SparseMat`

The cv::Mat Class: N-Dimensional Dense Array

⇒ Each matrix contains:

- flag ⇒ Signaling the contents of the array
- dims ⇒ Indicating the number of dimensions.
- rows ⇒ Indicating number of rows & Columns.
- ~~Columns~~ ⇒ Not valid for dims > 2
- cols
- data ⇒ pointer to where the array data is stored.
- refcount ⇒ reference counter analogous to the reference counter used by cv::Ptr<>.

⇒ The memory layout in data is described by the array step[].

⇒ An 2D Case

$$\&(mtx_{i,j}) = mtx.data + mtx.step[0]*i + mtx.step[1]*j$$

⇒ The data contained in cv::Mat is not required to be simple primitives.

Creating an Array

⇒ You can create an array simply by instantiating a variable of type cv::Mat.

⇒ An array created in this manner has no size and no data type.

⇒ You can, however, later ask it to allocate data by using a member function such as create().

⇒ One variation of Create() takes as arguments a number of rows, a number of columns and a type, and configures the array to represent a two-dimensional object.

⇒ The type of an array determines the kind of element it has as well as the number of channels.

CV_{8U, 16S, 16U, 32S, 32F, 64F}C{1, 2, 3}

→ Example ⇒ CV_32FC3

↓

{32-bit floating point three channel array.}

⇒ There are many constructors for cv::Mat, one of which takes the same arguments as Create() with an optional fourth element to initialize all of the elements in your new array.

```
cv::Mat m;
m.Create(3, 10, CV_32FC3);
m.SetTo(cv::Scalar(1.0f, 0.0f, 1.0f));
```

is equivalent to

```
cv::Mat m(3, 10, CV_32FC3, cv::Scalar(1.0f, 0.0f, 1.0f));
```

⇒ The data in an array is not attached rigidly to the array object.

↳ cv::Mat object is really a header for a data area, which - in principle - is an entirely separate thing.

⇒ The class cv::Mat also provides a number of static member functions to create certain kinds of commonly used arrays.

```
cv::Mat::zeros(rows, cols, type);
cv::Mat::ones(rows, cols, type);
cv::Mat::eye(rows, cols, type);.
```

* Accessing Array Elements Individually

⇒ There are several ways to access a matrix, all of which are designed to be convenient in different contexts.

⇒ The basic means of direct access is the (template) member function at<>.

Example

```
cv::Mat m = cv::Mat::eye(10, 10, 32FC1);
m.at<float>(3,3);
```

⇒ For a multichannel array:

```
cv::Mat m = cv::Mat::eye(10, 10, 32FC2);
m.at<cv::Vec2f>(3,3)[0];
m.at<cv::Vec2f>(3,3)[1];
```

1.0f));

⇒ You can also create array of complex number.

⇒ OpenCV provides a pair of iterator templates, one for const and one for non-const arrays.

- cv::MatIterator_<>
- cv::MatConstIterator_<>

Example

```
cv::Mat m(3, 52, CV_32FC3);
cv::MatConstIterator_<cv::Vec3f> it = m.begin();
while (it != m.end()) {
    (*it)[0]
    (*it)[1]
    (*it)[3]
    it++
}
```

★ The N-ary Array Iterator : NAryMatIterator

⇒ does not handle discontinuities in the packing of the arrays, but allows us to handle iteration over many arrays at once.

cv::NAryMatIterator

- Only requirement is that all of the arrays being iterated over be of the same geometry.
  (number of dimension & Extent in each dimension)

⇒ Instead of returning single elements of the array being iterated over, the N-ary iterator operates by returning chunk of those arrays, called **planes**.

plane
↳ Portion of the input array in which the data is guaranteed to be contiguous in memory.

Handling discontinuity

⇒ You are given contiguous chunk one by one.
↳ For each of such plane, you can either operate on it using array operations or iterate trivially over it yourself.

⇒ To initialize the cv::NAryMatIterator object, we need to have two things:

1. → C-style array containing pointers to all of the cv::Mats we wish to iterate over.
   ↳ Terminated by 0 or NULL.
2. → Another C-Style array of cv::Mats that can be used to refer to the individual planes as we iterate over them.

TODO

* Accessing Array Element by Block

{Accessing subset of an array as another array}

Table 4-7

→ {Methods for accessing array Elements by block}

→ When you use m.row() or m.col(), data in m is not copied to the new array.

⇒ Later, we will visit the copyTo() method, which actually will copy data.

⇒ The () operator is the only method to access that will allow you to extract a subvolume from a higher-dimensional array.

* Matrix Expressions

Table 4.8

m2 = m1

→ m2 will be another reference to data in m1

m2 = m1 + m0

→ m2 will have its separate memory.

* Saturation Casting

⇒ In OpenCV, you will often do operations that risk overflowing or underflowing the available values in the destination of some computation.

Example

```
Vxy = cv::saturate_cast<uchar>((Vxy-128)*2+128);
```

* More things an Array can do

Table 4-9

* The cv::SparseMat Class: Sparse Array

⇒ The cv::SparseMat class is used when an array is likely to be very large compared to number of nonzero entries.

⇒ The disadvantage of sparse representation is that computation with them is slower (on per-element basis).

⇒ The OpenCV sparse matrix class cv::SparseMat functions analogously to the dense matrix class cv::Mat in most ways.

↳ cv::SparseMat uses a hash table to store just the nonzero elements.

★ Accessing Sparse Array Elements

⇒ The most important difference between Sparse and dense arrays is how elements are accessed.

⇒ Sparse array provides four different access mechanisms:

- → cv::SparseMat::ptr()
- → cv::SparseMat::ref()
- → cv::Sparse.Mat::value()
- → cv::SparseMat::find()

5

Array Operations

Table 5-1

→ List of "friend" functions that either takes array types as arguments, have array type as return values or both