Robot Vision Systems

Lecture 2: Dense Matrices in OpenCV

Michael Felsberg

michael.felsberg@liu.se

OpenCV Types

- Before looking into matrices, some basic types (**classes**) need to be visited
- Several concepts are based on **templates**
- Several classes are based on **STL** (standard template library) **vectors**
- Seminar 1 will go into details about these terms

Primitive Datatypes

- (a tuple of) **unsigned char**, **bool**, **signed char**, **unsigned short**, **signed short**, **int**, **float**, **double**
- Identifier **CV_<bit-dpth>{U|S|F}C(<nm_chnls>)**
- Examples
  - uchar ~ CV_8UC1
  - 3-element floating-point tuple ~ CV_32FC3

Class **DataType**

- Template **trait** class
- Trait: A class used in place of template parameters. As a class, it aggregates useful types and constants
- Allows to get type information etc. from primitive types
- Example: DataType<float>::type

Point Classes

- Point_
  - 2D points
  - Point_<int> Point2i
  - Point2i Point
  - Point_<float> Point2f
  - Point_<double> Point2d
- Point3_
  - 3D points
  - Aliases Point3i, Point3f, Point3d

Small Matrices: **Matx**

- Type and size known at compilation time
- **Matx<float, R, C> MatxRCf**
- **Matx<double, R, C> MatxRCd**
- R,C = 1 .. 6
- Matx23f M(2, 3, 4,
  1, 0, -1);
- Access elements by M(r,c)
- Most matrix operations available
- If not, convert to general matrices (and back)

Small Vectors: **Vec**

- Column vectors (C=1) as special case of Matx
- **Vec<uchar, R> VecRb**
- **Vec<short, R> VecRs**
- **Vec<int, R> VecRi**
- **Vec<float, R> VecRf**
- **Vec<double, R> VecRd**
- R = 2..6
- Access by [r]
- Conversion of Vec<T, 2/3/4> to Point_, Point3_, Scalar_
- **Scalar_<double> Scalar**

Smart Pointers: **Ptr**

- Template class for wrapping pointers
- Similar to std::shared_ptr from C++11
- Avoids copying data, just generates additional headers
- Reference counting, for C++ classes, fully automatic deallocation
- Thread-safe
- Advanced use: can be applied to base-classes

General Matrices: **Mat**

- Multi-dimensional dense array class
- Can be used to store (more or less) all data:
  - real or complex-valued vectors and matrices
  - grayscale or color images
  - voxel volumes
  - vector fields
  - point clouds
  - tensors
  - histograms

Memory Arrangement (2D)

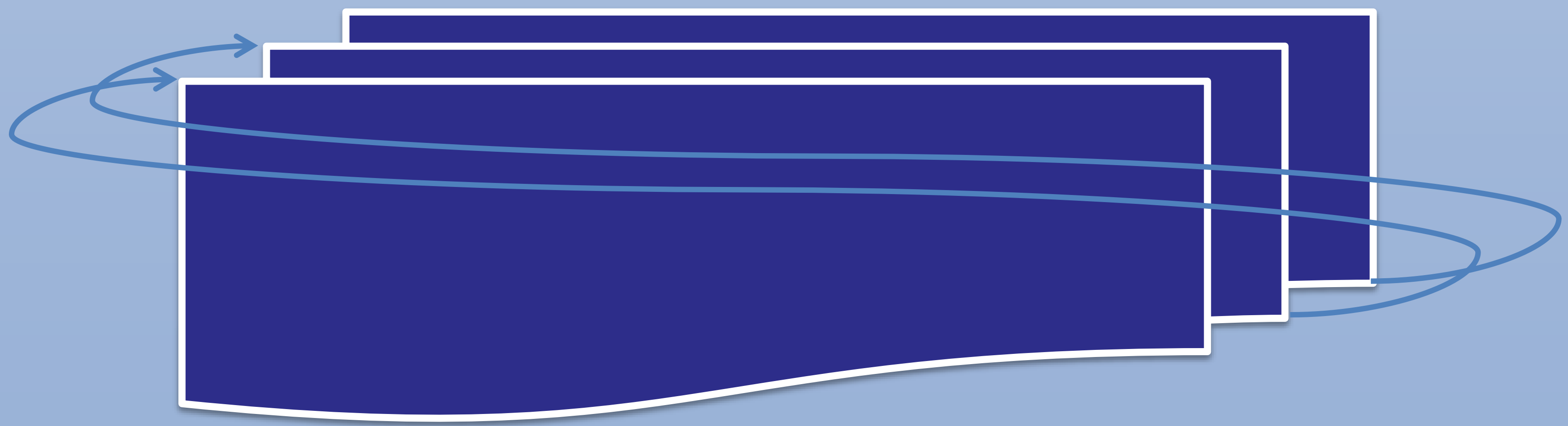
- Array M.step[] defines address calculation: addr(M_{r,c}) = M.data + M.step[0]*r + M.step[1]*c
- M.step[0] >= M.step[1]*M.size[1]
- Stored row-by-row
- M.step[1] = M.elemSize()

M.data	M.data+M.elemSize()	M.data+2*M.elemSize()	M.data+3*M.elemSize()
M.data+4*M.elemSize()	M.data+5*M.elemSize()	M.data+6*M.elemSize()	M.data+7*M.elemSize()
M.data+8*M.elemSize()	M.data+9*M.elemSize()	M.data+10*M.elemSize()	M.data+11*M.elemSize()

M.step[0] = 4*elemSize()

Memory Arrangement (nD)

- addr(M_{i_0,...,i_{M.dims-1}}) = M.data +
  M.step[0]*i_0 + M.step[1]*i_1 + ...
  + M.step[M.dims-1]*i_{M.dims-1}
- M.step[i] >= M.step[i+1]*M.size[i+1]
- M.step[M.dims-1] = M.elemSize() is minimal
- 3D array: plane-by-plane



Creating Matrices

- 2D: **create(R,C,type) / Mat(R,C,type[,value])**
  - Mat M2(2,3,CV_32FC2,Scalar(0,1));
- 3D: **Mat(dims,sizes,type[,value])**
  - int sz[] = {2,3,2}; Mat M3(3, sz, CV_8U, Scalar:all(0));
- Copy constructor (smart Ptr!) or Mat::clone()
- Header for user data Mat(R,C,type,ptr[,step])
  - double m[2][2] = {{2,3},{1,0}};
  - Mat M = Mat(2,2,CV_64F,m);
- Initializers:
  - M += Mat::eye(M.rows,M.cols,CV_64F);
  - Mat M4 = (Mat_<double>(2,2) << 2,3,1,0);

Useful Types

- **Size_** class for size of image or rectangle
  - **Size_<int> Size2i**
  - **Size2i Size**
  - **Size_<float> Size2f**
- **Range r** contains r.start and r.end
  - Range(a,b) translates to a:b-1 in Matlab and a..b in Python
  - Range::all() translates to : in Matlab and … in Python

Rectangles

- **Rect_** class for 2D rectangles
  - Top-left corner: Rect_::x, Rect_::y
  - height and width (right and bottom boundary excluded)
  - **Rect_<int> Rect**
  - Use for ROIs
- **M.row(r)** / **M.col(c)**: select row r / column c
  - A.row(i) = A.row(j) **+ 0**;
- **M.rowRange(r,h)** / **M.colRange(c,w)**: select range of rows r..r+h-1 / columns c..c+w-1

Constructors

- Mat::Mat()
- Mat::Mat(int **rows**, int **cols**, int **type**)
- Mat::Mat(Size **size**, int **type**)
- Mat::Mat(int **rows**, int **cols**, int **type**, const Scalar& **s**)
- Mat::Mat(Size **size**, int **type**, const Scalar& **s**)
- Mat::Mat(const Mat& **m**)
- Mat::Mat(int **rows**, int **cols**, int **type**, void* **data**, size_t **step**=AUTO_STEP)

Constructors

- Mat::Mat(Size **size**, int **type**, void* **data**, size_t **step**=AUTO_STEP)
- Mat::Mat(const Mat& **m**, const Range& **rowRange**, const Range& **colRange**=Range::all())
- Mat::Mat(const Mat& **m**, const Rect& **roi**)
- Mat::Mat(const CvMat* **m**, bool **copyData**=false)
- Mat::Mat(const IplImage* **img**, bool **copyData**=false)

use cvarrToMat() instead

Constructors

- Mat::Mat(const Vec<T, n>& **vec**, bool **copyData**=true)
- Mat::Mat(const Matx<T, m, n>& **vec**, bool **copyData**=true)
- Mat::Mat(const std::vector<T>& **vec**, bool **copyData**=false)
- Mat::Mat(int **ndims**, const int* **sizes**, int **type**)
- Mat::Mat(int **ndims**, const int* **sizes**, int **type**, const Scalar& **s**)

Constructors

- Mat::Mat(int **ndims**, const int* **sizes**, int **type**, void* **data**, const size_t* **steps**=0)
- Mat::Mat(const Mat& **m**, const Range* **ranges**)
- Mat::Mat(const MatCommaInitializer_<T> & **commaInitializer**) (see page 12)
- Mat::Mat (const cuda::GpuMat & **m**)

Element Access

- Single element **M.at<double>(r,c)** (slow)
- Single row **const double* Mi = M.ptr<double>(r);** (faster)
- Whole matrix as one row (requires **M.isContinuous()**): r=0 (fastest)
- Iterator **MatConstIterator_<double> it = M.begin<double>(), it_end = M.end<double>(); for(; it != it_end; ++it) fun(*it)** (fast)

Undocumented: range-based for loops (C++11)

Multichannel Matrices

- If a matrix is of multichannel type (CV_<bit-dpth>{U|S|F}C(<nm_chnls>) with nm_chnls>1)
  - Access single channel in single element as M.at<double>(r,c)[k] / (*it)[k]
  - elemSize() is k*sizeof(double)
  - Example: r=3, c=2, k=2

M.data	M.data+sizeof(double)	M.data+M.elemSize()	M.data+M.elemSize()+sizeof(double)
M.data+2*M.elemSize()	M.data+2*M.elemSize()+sizeof(double)	M.data+3*M.elemSize()	M.data+3*M.elemSize()+sizeof(double)
M.data+4*M.elemSize()	M.data+4*M.elemSize()+sizeof(double)	M.data+5*M.elemSize()	M.data+5*M.elemSize()+sizeof(double)

Template Mat_

- Template class derived from Mat
- More convenient if many accesses and type known
  - Mat_<Vec3b> img(..);
  - img(r,c) = Vec3b(0,255,255);

Generic Arrays

- Only used for own functions with unknown in-/output array
- Stands for Mat, Mat_, Matx, std::vector<T>
- **InputArray** for input
  - getMat() constructs header
  - kind() distinguishes Mat and vector<>
- **OutputArray** for output with additional
  - create() (to be called before getMat())
  - needed() checks whether output required (noArray())

Elementary Methods

- Methods implementing (computational) functionalities: next lecture
- Already mentioned
  - Initializers Mat::**eye**(R,C,T), Mat::**eye**(size,T), Mat::**zeros**(), Mat::**zeros**(dims, sizes, T), Mat::**ones**()
  - Rows/columns Mat::**row**(r), Mat::**col**(c)
  - Row-/columnranges Mat::**rowRange**(start,end), Mat::**rowRange**(range), Mat::**colRange**()
  - Mat::**clone**()

Further Methods

- Assignment:
  - –Mat::**operator=**(Mat&) (no copy)
  - –Mat::**operator=**(MatExpr&) (smart allocation)
  - –Mat::**operator=**(Scalar&) (each element assigned)
- Mat::**copyTo**(OutputArray[, InputArray]) use this instead of 1st assignment for enforcing copy; a mask can be specified
- Mat::**setTo**(InputArray[, InputArray]) advanced variant of 3rd assignment