Robot Vision Systems

Lecture 3: Methods for Dense Matrices in OpenCV

Michael Felsberg

michael.felsberg@liu.se

Further Methods

- Mat::**diag**(int d=0) (d<0 upper, d>0 lower half)
- Mat::**convertTo**(OutputArray,int T[,double a, b) convert to different type (T), after scaling by a and adding b
- Mat::**assignTo**(Mat&[, int T]) functional form of convertTo
- Mat::**reshape**(int k, int r=0) changes channels to k (0: no change) and rows to r (att: for new r, matrix must be continuous)

Excerpt: Reshape

- Very important = useful (i)Python example
  - import cv2.cv as cv
  - capture = cv.CaptureFromCAM(0)
  - img = cv.QueryFrame(capture)
  - A = cv.GetMat(img)
  - B = cv.CloneMat(A)
  - C = cv.Reshape(A,1)
  - C = cv.Reshape(A,0,512)
  - C = cv.Reshape(B,0,512)
  - cv.NamedWindow("camera", 1)
  - cv.ShowImage("camera", A)
  - cv.WaitKey(1)
  - cv.DestroyAllWindows()

Further Methods

- Explicit allocation
  - Mat::**create**(int R, int C, int T), Mat::**create**(Size size, int T) 2D matrices
  - Mat::**create**(int N, const int* sizes, int T) ND matrices
- Explicit handling of references (avoid!)
  - Mat::**addref**() increase of counter
  - Mat::**release**() decrease + deallocation if necessary
- Mat::**resize**(size_t R[, const Scalar& s]) resize to R rows, potentially fill with s
- Mat::**reserve**(size_t R) reserve R rows

Further Methods

- Using matrix as list
  - Mat::**push_back**(const T& elem)
  - Mat::**push_back**(const Mat& m)
  - Mat::**pop_back**(size_t nelems=1)
- ROI handling
  - Mat::**locateROI**(Size& size, Point& offs) returns offset and size of current ROI in embedding matrix
  - Mat::**adjustROI**(int dt, int db, int dl, int dr) changes ROI boundaries by respective values

Further Methods

- ROIs
  - Mat::**operator()** (Range, Range)
  - Mat::**operator()** (Rect&)
  - Mat::**operator()** (const Range*)
- Mat::**total()** number of elements (pixels)
- Mat::**channels()** number of channels
- Mat::**elemSize1()** elemSize()/channels()
- Mat::**step1(int)** step divided by elemSize1()
- Mat::**type()** type of elements
- Mat::**depth()** type/bit-depth (per channel)

Further Methods

- Mat::**size()** 2D matrix size
- Mat::**empty()** true if total()=0 or data=NULL
- Mat::**at(int[, int, int])** 1D (2D, 3D) access
- Mat::**at(Point)** 2D access
- Mat::**at(const int*)** ND access
- Mat::**begin()** iterator start
- Mat::**end()** iterator end

NAryMatIterator

- **NAryMatIterator**(const Mat** arrays, Mat* planes, int narrays=-1)
  - Element-wise operations on N multi-D arrays
  - Same geometry (dimensionality & sizes)
  - it.planes[0..N-1] are the (continuous!) slices of the corresponding matrices 0..N-1
- Typical example
  - NAryMatIterator it(&arrays, &planes, N);
  - for(int p = 0; p < it.nplanes; p++, ++it)

    {.. it.planes[n] ..}

Matrix Expressions

- Mat A,B; Scalar s; double alpha
  - Addition etc: A+B, A-B, A+s, A-s, s+A, s-A, -A
  - Scalar multiplication: A*alpha
  - Per-element (Hadamard) multiplication and division: A.mul(B,[alpha]), A/B, alpha/A
  - Matrix multiplication: A*B
  - Transposition: A.t()

Inversion

- (Pseudo) inversion, solving linear systems, LS: A.inv([method]), A.inv([method])*B
- Method:
  - DECOMP_LU: standard, LU-decomposition for non-singular matrices
  - DECOMP_CHOLESKY: symmetrical, positive definite matrices; 2x faster than LU
  - DECOMP_SVD: PI for singular / non-square matrices
  - Further: QR, EIG
  - DECOMP_NORMAL: normal equations

Matrix Expressions

- Per-element comparison: >, >=, ==, !=, <=, < (between A, B, alpha)
- Bitwise logical: ~, &, |, ^ (between A, B, s)
- Element-wise min(), max() (for A, B, alpha)
- Element-wise abs(A)
- Cross- and dot-product (Frobenius) A.cross(B), A.dot(B)
- norm(), mean(), sum(), countNonZero(), trace(), determinant(), repeat()

Operations on Arrays

- We omit operations that are more convenient to write as matrix expressions (e.g. add(), addWeighted(), bitwise_YYY(), compare(), divide(), gemm(), transpose() …)
- We omit very specific operations (calcCovarMatrix(), cartToPolar(), getConvertElem(), getOptimalDFTSize(), …)
- Useful general operations:
  - Element-wise absdiff(A,B,C), works also with scalars
  - checkRange(A[, quiet=true, …]) returns false / throws an exception if an element is NaN or inf
  - completeSymm(A[, lowerToUpper=false): copies upper half to lower (depending on flag)

Operations on Arrays

- convertScaleAbs(src, dst, alpha=1, beta=0): useful to generate image (dst, 8 bit) from a matrix src by affine mapping + absolute value
- dct/dft(src, dst, flags=0): integral transforms
  - DCT/DFT_INVERSE flag
  - DCT/DFT_ROWS flag (row-wise 1D transforms)
  - DFT_SCALE divide by number of elements
  - DFT_COMPLEX_OUTPUT expand to full hermitian symmetry; usually packed real
  - DFT_REAL_OUTPUT assumes hermitian symmetry before inverse DFT

Operations on Arrays

- **eigen**(InputArray **src**, OutputArray **eigenvalues**[, OutputArray **eigenvectors**])
  - input matrix CV_32FC1 or CV_64FC1 type
  - square size
  - Symmetrical
- **exp**(InputArray **src**, OutputArray **dst**)
  - Point-wise exponential
  - NaN, Inf not handled
- Question: how to write mexp()?

Operations on Arrays

- pow(InputArray **src**, double **power**, OutputArray **dst**)
  - –Non-integer powers: abs(src) used (!)
- sqrt(InputArray **src**, OutputArray **dst**)
- log(InputArray **src**, OutputArray **dst**)
- phase(InputArray **x**, InputArray **y**, OutputArray **angle**, bool **angleInDegrees**=false)
  - –angle = atan2(y,x)
- magnitude(InputArray **x**, InputArray **y**, OutputArray **magnitude**)