Robot Vision Systems

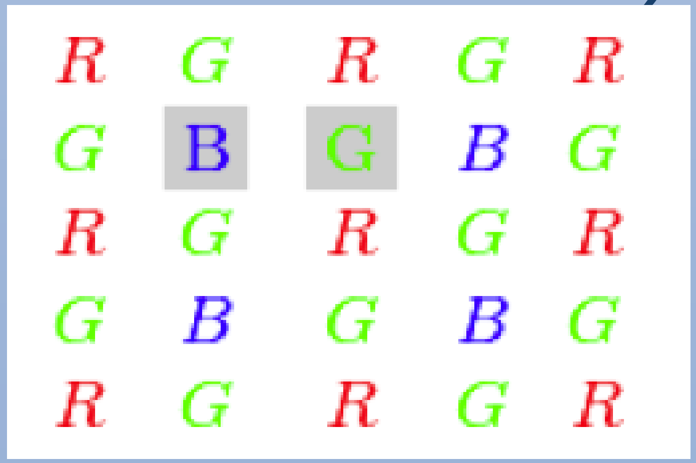
Lecture 4: Sparse Matrices in OpenCV

Michael Felsberg

michael.felsberg@liu.se

Operations on Arrays

- split(const Mat& **src**, Mat* **mvbegin**)
  split(InputArray **m**, OutputArrayOfArrays **mv**)
- merge(const Mat* **mv**, size_t **cnt**, OutputArray **dst**)
  merge(InputArrayOfArrays **mv**, OutputArray **dst**)
- cvtColor(InputArray **src**, OutputArray **dst**, int **code**, int **dstCn**=0)
  - Code (enum **ColorConversionCodes**: RGB – BGR, alpha, 16bit, edge-aware Bayer, but not colornames)
    CV_RGB2GRAY, CV_BGR2Luv, CV_BGR2XYZ, CV_BGR2YCrCb, CV_BGR2HSV, CV_BGR2HLS, CV_BGR2Lab, CV_BayerBG2BGR



Operations on Arrays

- mixChannels(const Mat* **src**, size_t **nsrcs**, Mat* **dst**, size_t **ndsts**, const int* **fromTo**, size_t **npairs**)
  mixChannels(InputArrayOfArrays **src**, InputOutput\ ArrayOfArrays **dst**, const int* **fromTo**, size_t **npairs**)
  mixChannels(InputArrayOfArrays **src**, InputOutput\ ArrayOfArrays **dst**, const vector<int>& **fromTo**)
- split RGBA into BGR + separate alpha

  ```
  Mat rgba( 100, 100, CV_8UC4, Scalar(1,2,3,4) );
  Mat bgr( rgba.rows, rgba.cols, CV_8UC3 );
  Mat alpha( rgba.rows, rgba.cols, CV_8UC1 );
  Mat out[] = { bgr, alpha };
  int from_to[] = { 0,2, 1,1, 2,0, 3,3 };
  mixChannels( &rgba, 1, out, 2, from_to, 4 );
  ```
- NEEDS PRE-ALLOCATION!

Operations on Arrays

- insertChannel(InputArray **src**, InputOutputArray **dst**, int coi)
  - inserts a single channel at coi (undocumented)
- flip(InputArray **src**, OutputArray **dst**, int **flipCode**)
  - flipCode =0: x-axis, >0: y-axis, <0: both
- inRange(InputArray **src**, InputArray **lowerb**, InputArray **upperb**, OutputArray **dst**)
  - lowerb <= src <= upperb
  - bounds may be arrays or scalars
- LUT(InputArray **src**, InputArray **lut**, OutputArray **dst**)
  - Look-up table, works also for multiple channels
  - src is CV_8; if signed, LUT adds 128

Operations on Arrays

- minMaxIdx(InputArray **src**, double* **minVal**, double* **maxVal**=0, int* **minIdx**=0, int* **maxIdx**=0, InputArray **mask**=noArray())
  - –finds ND-index and value of minimum / maximum
  - –index is at least 2D
  - –mask can be used
  - –single channel
- minMaxLoc(InputArray **src**, double* **minVal**, double* **maxVal**=0, Point* **minLoc**=0, Point* **maxLoc**=0, InputArray **mask**=noArray())
  - –Similar, but 2D (Point*)

Operations on Arrays

- normalize(InputArray **src**, InputOutputArray **dst**, double **alpha**=1, double **beta**=0, int **norm_type**=NORM_L2, int **dtype**=-1, InputArray **mask**=noArray())
  - norm_type may be NORM_INF, NORM_L1, NORM_L2, NORM_MINMAX
  - normalizes to alpha (cases 1-3)
  - normalizes range to [alpha,beta] (case 4)

Operations on Arrays

- reduce(InputArray **src**, OutputArray **dst**, int **dim**, int **rtype**, int **dtype**=-1)
  - dim determines direction of reduction: 0 - to a single row; 1 – to a single column
  - rtype determines type of reduction: CV_REDUCE_SUM, CV_REDUCE_AVG, CV_REDUCE_MAX, CV_REDUCE_MIN
- repeat(InputArray **src**, int **ny**, int **nx**, OutputArray **dst**)
- Mat repeat(const Mat& **src**, int **ny**, int **nx**)

Operations on Arrays

- solve(InputArray **src1**, InputArray **src2**, OutputArray **dst**, int **flags**=DECOMP_LU)
  - Other methods: DECOMP_CHOLESKY, DECOMP_EIG, DECOMP_SVD*, DECOMP_QR*
  - Flag DECOMP_NORMAL* means to solve src1.t()*src1*dst=src1.t()*src2 instead of src1*dst=src2
  - Methods with * can also be used to solve LS problem
  - For unity-norm systems (eg motion tensor), use SVD::solveZ() instead

Operations on Arrays

- solveCubic(InputArray **coeffs**, OutputArray **roots**) – coeffs may have 3 or 4 coefficients
- solvePoly(InputArray **coeffs**, OutputArray **roots**, int **maxIters**=300) – iterative computation of roots
- sort(InputArray **src**, OutputArray **dst**, int **flags**)
- sortIdx(InputArray **src**, OutputArray **dst**, int **flags**)
  - Flags: CV_SORT_EVERY_ROW/COLUMN, CV_SORT_ASCENDING/DESCENDING

Classes

- Special classes implement relevant functionalities:
  - PCA (principal components)
  - RNG (random numbers) – second version available (Mersenne Twister)
  - SVD (singular values)
  - LDA (linear discriminant analysis) - undocumented

SVD

- Example pseudo inverse
  - SVD svd(A)
  - Mat pinvA = svd.vt.t()*Mat::diag(1./svd.w)*svd.u.t();
- Example LM
  - X -= (A.t()*A + lambda*Mat::eye(A.cols,A.cols,A.type())).inv(DECOMP_CHOLESKY)*(A.t()*err);

Transforms

- transform()
- perspectiveTransform()
- getAffineTransform()
- getPerspectiveTransform()
- estimateRigidTransform()
- findEssentialMat()
- findFundamentalMat()
- findHomography()
- warpAffine()
- warpPerspective()

Sparse Matrices

- Classes SparseMat and SparseMat_
  - A hash table of nodes: struct with fields hashval, next, and idx[]
  - Standard constructors + copy from Mat
  - SparseMat(int dims, const int* sizes, int type)
  - Standard assignment operators, also from Mat
  - clone(), copyTo(), convertTo() work as expected, latter two also for Mat
  - create(), clear(), addref(), release() as expected

Sparse Matrices

- Methods on structure
  - elemSize() element size in byte (not including node)
  - elemSize1(), type(), depth(), channels() as in Mat
  - size(), size(i), dims(), nzcount() as expected
- Hash element value: hash(int[, int, int]) or hash(const int*)
- Element access by ref() (returns reference), value() (returns value), find() (returns pointer)
  - argument same as hash(), optional hash-value
- erase() for erasing elements

Sparse Matrices

- Iterators
  - SparseMatIterator
  - SparseMatConstIterator
- Access value in node
  - value(Node* n)
  - const value(const Node* n) const
  - Node can be obtained from iterator (Node* n= it.node())
- Special case SparseMat_: instead of value(), use just “()”

Operations on Arrays

- minMaxLoc(const SparseMat& **a**, double* **minVal**, double* **maxVal**, int* **minIdx**=0, int* **maxIdx**=0)
- norm(const SparseMat& **src**, int **normType**)
- normalize(const SparseMat& **src**, SparseMat& **dst**, double **alpha**, int **normType**)

Differences

- Major differences using sparse matrices are
  - Element access is NOT done using at(), but value() or ref()
  - There is no NArySparseMatIterator, but you can iterate through nodes na of matrix A and access elements in B as B.value(na->idx,&na->hashval)
  - There are hardly any arithmetic operations on sparse matrices, in particular no matrix product
  - This will be topic of the exercise on Thursday!

Other Comments

- Maximum number of channels
  - Documentation says it is CV_MAX_CN = 32
  - Actually it is CV_CN_MAX = 512
- Datatype Scalar_()
  - may be initialized with any number of elements between 1 and 4
  - can be assigned to any other type up to 4 channels
  - Some operations might only work up to 4 channels (mean(), randn(), sum())