The VIGRA Image Analysis Library

Ullrich Köthe

June 2012

Multidimensional Image Processing
Heidelberg Collaboratory for Image Processing
University of Heidelberg

Outline

- Goals and History
- Philosophy
- Important Abstractions
- Contents
- Usage Examples
- Outlook

Goals and History

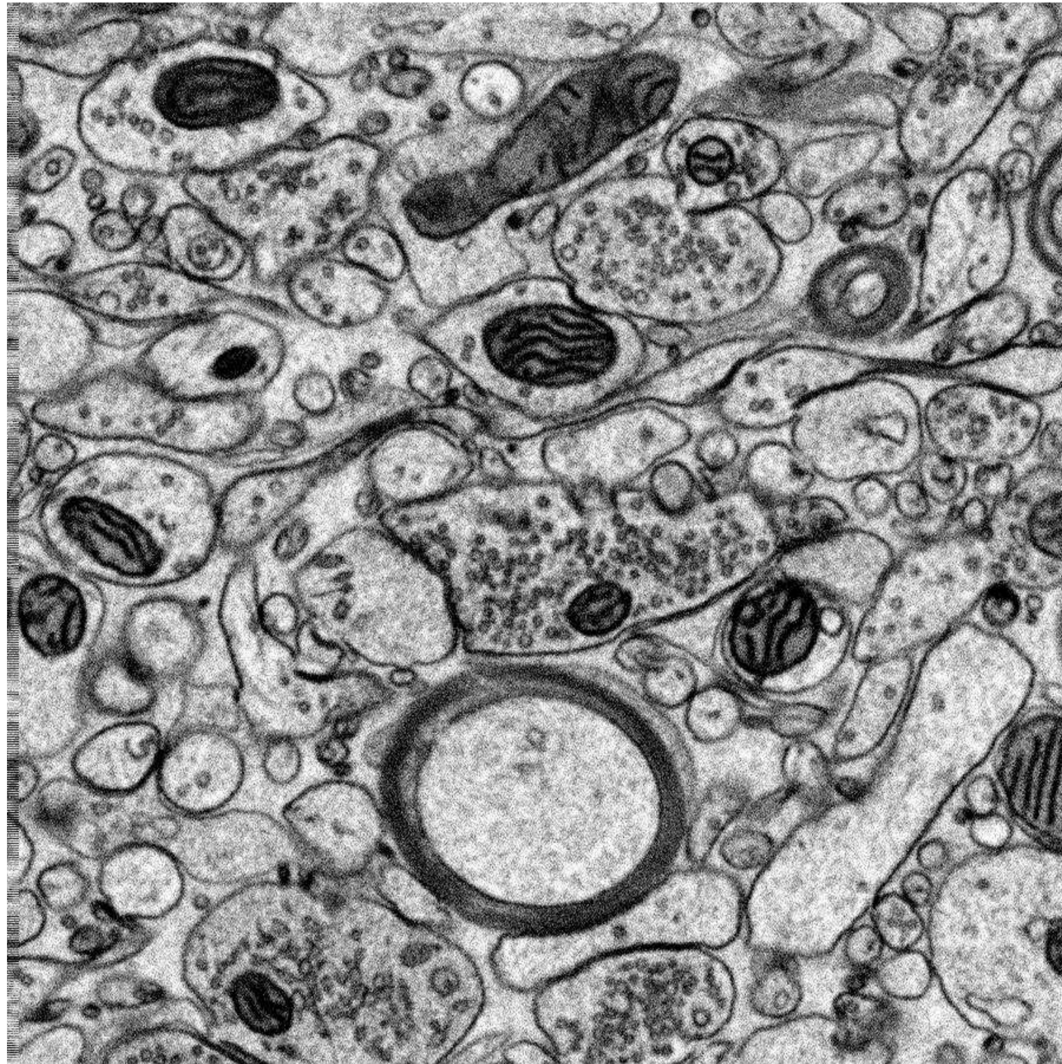
- Started in 1997 as part of U.Köthe's PhD project
  - original goal: generalize the C++ STL iterators to 2D
  - implement generic image processing and analysis algorithms
  - define efficient abstract interfaces for image analysis
- First official release in 2000 under MIT license
- Continuous evolution
  - 2002: impex library (import/export of common image formats)
  - 2003: multi-dimensional arrays and algorithms
  - 2004: numerics (linear algebra, polynomial solvers)
  - 2005: SplineImageView (transparent on-demand interpolation)
  - 2009: machine learning (random forest)
  - 2010: vigranumpy (Python bindings, numpy compatible), ilastik GUI, CMake-based build system
  - 2012: generic object statistics (accumulators)
- Lots of unofficial functionality

Philosophy

- Fundamental algorithms for *arbitrary* dimensions in *generic* form
  - Abstractions for higher-level algorithm design
    - model the application domain
    - standardize algorithm <==> data structure interface
  - Templates in C++
    - eliminate abstraction overhead (continually improving!)
    - work for very large datasets
    - configure automatically for many use cases
  - Scripting language bindings (Python, Matlab)
    - rapid prototyping
    - full integration with numpy (no data conversion or replication!)
    - Python-level parallelization
  - High quality
    - algorithms that stood the test of time (not: as many as possible)
    - extensive test suite (30% of the code)
    - code readability and ease-of-use
  - Portability (Linux, Windows, MacOS X, 32- and 64-bit)

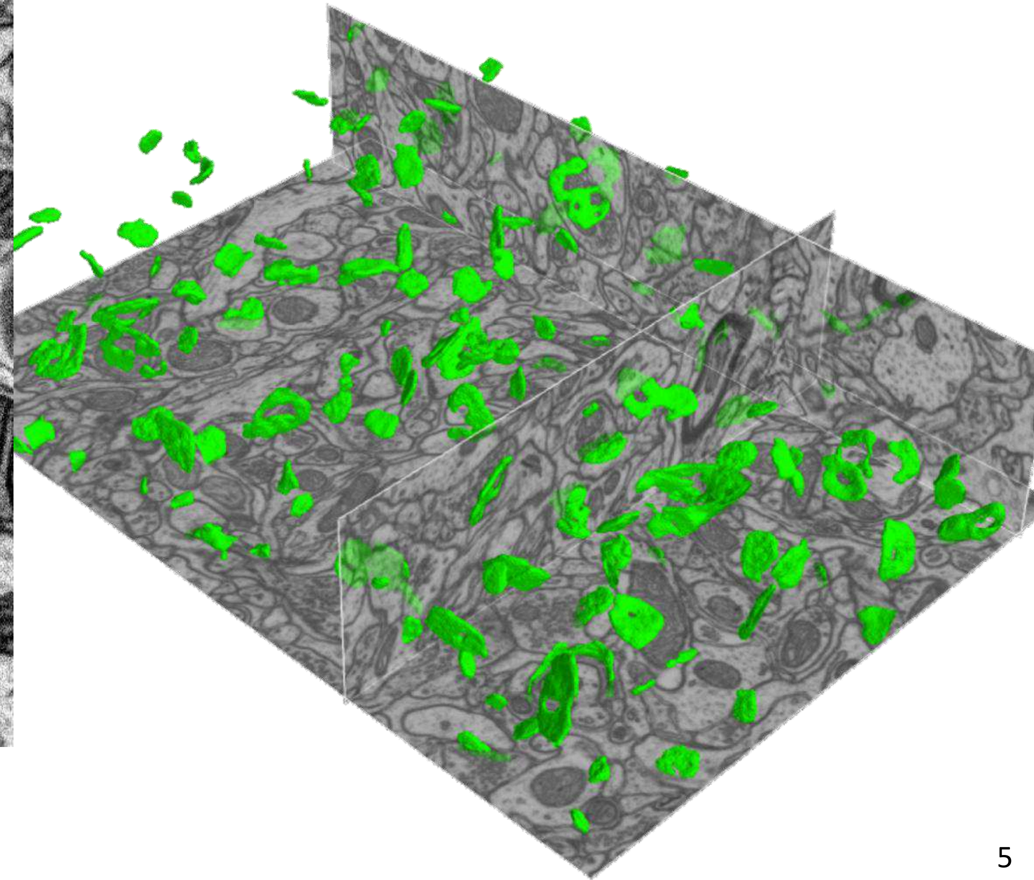
Example Use: Connectomics

How is the brain wired?



3D electron microscopy of neural tissue ($\approx 2000^3$ @ 5nm = 8 billion voxels, data by G. Knott et al. 2010)

Example: Synapse detection:



Images and 2-dimensional Iterators

- act on x- and y-axis independently

```
typedef  BasicImage<UInt8>  Image;
typedef  Image::traverser   Iterator;

Image  image(800, 600);

Iterator  end = image.lowerRight();
int count = 0;
for(Iterator iy = image.upperLeft();  iy.y < end.y;  ++i.y)
{
    for(Iterator ix = iy;  ix.x < end.x;  ++ix.x)
    {
        *ix = ++count;
    }
}
```

- change ROI by moving iterators to ROI corners:

```
image.upperLeft()+Diff2D(6,5), image.lowerRight()-Diff2D(5,4)
```

Multi-dimensional Arrays and Views

- resembles high-level syntax of Matlab
- works in arbitrary dimensions by recursion through binding
- crucial for VIGRA multi-dimensional algorithms

```
typedef  MultiArray<3, double>      Array;
typedef  MultiArrayView<3, double>  View3;
typedef  MultiArrayView<3, double>  View2;
typedef  Array::difference_type     Shape;

Array  array(Shape(40, 30, 20));    // automatic zero-initializ.

array(2,3,4) = array[Shape(7,8,9)]; // access to elements
Array  b = sqrt(array) + 4.0;       // expression templates

View3 sub = array.subarray(Shape(3,4,5), Shape(34,23,16));
View3 zxy = array.transpose(Shape(2,0,1)); // change index order

View2 xy = array.bindOuter(5);             // fix z=5
View2 xz = array.bind<1>(10);              // fix y=10
```

Linear Iteration on Multi-Dimensional Arrays

- became recently feasible due to compiler/processor evolution
  - no abstraction overhead despite possible strides
  - scan order for current subarray and current index order (from first to last index)

```
array.begin(), array.end()    // iterator pair like in STL
                              // default scan order = memory order

array.subarray(shape1, shape2).begin()  // scan over subarray only
array.transpose().begin()    // scan over z-dimension first
array.bindInner(2).begin()   // scan over y and z ( x=2 is fixed)
array.bindOuter(5).begin()   // scan over x and y ( z=5 is fixed)
```

- simultaneous iteration over several arrays

```
auto i = createCoupledIterator(array1, array2, array3);
i.get<1>(), i.get<2>(), i.get<3>()   // access current elements
i.point()                            // access current coordinates
```

Type Inference and Reflection

- Traits classes
  - derive types of intermediate variables and end results

```
typedef PromoteTraits<InType1, InType2>::Promote  OutType;
NormTraits<Type>::NormType norm = array.norm();
```

  - control type conversion between Python and VIGRA

```
NumpyArrayTraits<3, Type>::compatible((PyArrayObject *)o);
```

  - get crucial information about types
    - smallest and largest element:

```
NumericTraits<Type>::min();
NumericTraits<Type>::max();
```

    - convert from floating-point representation to `Type`, possibly with rounding and clamping:

```
Type v = NumericTraits<Type>::fromRealPromote(rv);
```

  - work uniformly for scalar, vector, and array types
  - may become less crucial in C++11 due to new `auto` and `decltype` keywords

Option Objects

- Flexible algorithms have many options
  - Most options are left at default values
  - C++ arguments allow defaults only at the end
- Explicitly specify arbitrary option subsets by option objects
  - Inspired by Python keyword arguments

```
gaussianSmoothMultiArray(multiArrayRange(a), multiArray(b), scale,
                ConvolutionOptions<3>()
                   .filterWindowSize(2.0) // window is 2*scale
                   .stepSize(1, 1, 3.2)   // z resolution is lower
                   .subarray(Shape3(40,40,10), Shape3(200,60,40)));
```

Library Contents

- Array data structures (n-dimensional)
  - expression templates for easy array algebra and arithmetic
  - file I/O (image file formats, HDF5 for multi-dimensional and structured data)
- Filters
  - convolution, resize, morphology, distance transform, Fourier transform in nD
  - non-linear diffusion and total variation in 2D
- Features
  - Gabor filter banks, boundary and energy tensor
  - differential n-jets (eigenvalues of Hessian matrix are very popular)
- Image Analysis
  - Edge and corner detection
  - Region growing and watersheds
  - Object statistics
- Machine learning (classification, regression, e.g. Random Forest)
- Numerics
  - linear algebra (linear solvers, symmetric/unsymmetric eigen decomposition)
  - least squares (linear, non-linear, ridge regression, LASSO)

Example: Data Import and Export

- import and export an image

```
ImageImportInfo info("lenna.png");
Shape2 shape(info.width(), info.height());
MultiArray<2, RGBValue<UInt8> > image(shape);
importImage(info, destImage(image));

exportImage(srcImageRange(image, RGBToGrayAccessor()),
            ImageExportInfo("lenna_gray.jpg"));
```

- import volume data as a whole or in part

```
HDF5File datafile("volume_data.h5", HDF5File::Open);

MultiArray<3, float> volume;
datafile.readAndResize("data", volume);

Shape3 blockShape = volume.shape() / 2, blockOffset(5,10,20);
MultiArray<3, float> block(blockShape);
datafile.readBlock("data", blockOffset, blockShape, block);
```

Example: Watersheds

```
MultiArray<2, float> input(...);

MultiArray<2, float> gradient(input.shape());

double scale = 2.0;
gaussianGradientMagnitude(srcImageRange(input),
                          destImage(gradient),
                          scale);

MultiArray<2, int> labels(input.shape());
generateWatershedSeeds(srcImageRange(gradient),
                       destImage(labels), // seeds: minima below 2.0
                       SeedOptions().minima().threshold(2.0));

watershedsRegionGrowing(srcImageRange(gradient),
                        destImage(labels), //seeds will be overwritten
                        FourNeighborCode(),// use 4-neighborhood
                        WatershedOptions().completeGrow());
                               // use interpixel boundaries
```

Example: Random Forest

- ensemble of randomized decision trees
- fast, easy to train, low error rate

```
int n = 200;  // number of training examples
int m = 3;    // number of features

Matrix<float> training_features(n, m), true_labels(n, 1);
... // put training data into feature and label matrices

RandomForest<float> rf(RandomForestOptions().tree_count(100));
rf.learn(training_features, training_labels); // train classifier

int N = ...;  // number of samples for prediction
Matrix<float> features(N, m),
              class_probabilities(N, rf.class_count());
... // compute features
rf.predictProbabilities(features, class_probabilities);
```

Example: Random Forest Prediction on an Image

```
int width = input.shape(0), height = input.shape(1);
MultiArray<3, float> feature_image(Shape3(width, height, 3));

// compute three features
feature_image.bindOuter(0) = input;     // raw input as feature 0
gaussianSmoothing(srcImageRange(input),
                  destImage(feature_image.bindOuter(1), scale);
gaussianGradientMagnitude(srcImageRange(input),
                  destImage(feature_image.bindOuter(2), scale);

MultiArrayView<2, float> features =
                  feature_image.asShape(Shape2(width*height, 3));

MultiArray<2, int> label_image(Shape2(width, height));
MultiArrayView<2, int> labels =
                  label_image.asShape(Shape2(width*height, 1));

rf.predictLabels(features, labels);
```

Example: Configurable Statistics via Accumulators

- Statistics are easy to compute, but there is a combinatorial explosion of possibilities (global or per region, on values or on coordinates, weighted or unweighted, plain or centralized or normalized, ...)
- Generic creation of desired set via AccumulatorChain:

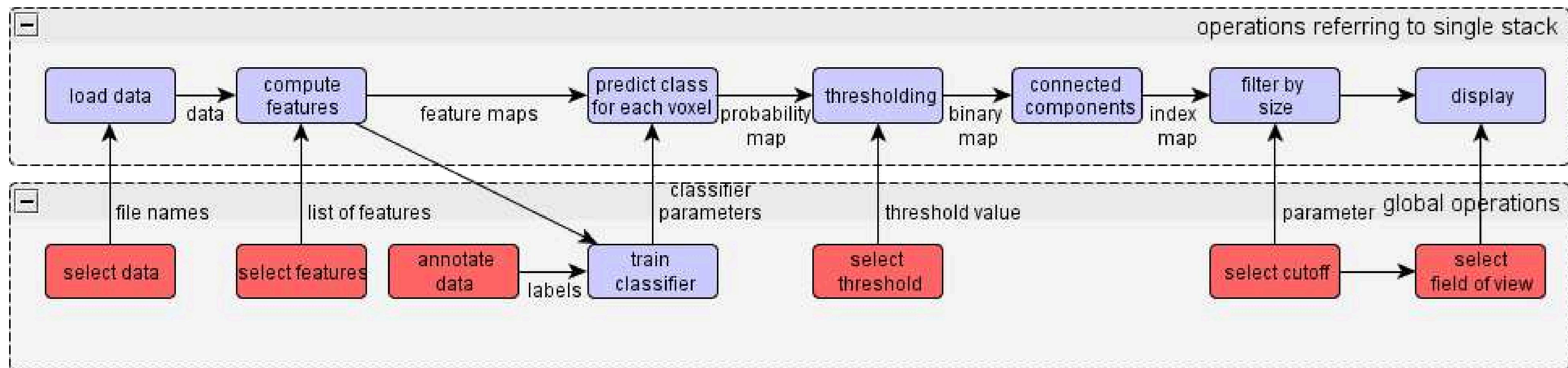
```
MultiArray<3, double>  data(…);
MultiArray<3, UInt32> labels(…);
typedef  CoupledIteratorType<3, double, UInt32>::type  Iterator;

AccumlatorChainArray< Iterator::value_type,
   Select<DataArg<1>, LabelArg<2>,
          Mean, Variance,     // per-region statistics over values
          Coord<Mean>, Coord<Variance> ,  // and over coordinates,
          Global<Mean>, Global<Variance>>>  // global statistics
a;

Iterator start = createCoupledIterator(data, labels),
           end = start.getEndIterator();
collectStatistics(start, end, a);
```

Universität Heidelberg

HCI

Python-Level Parallelization

- Example workflow: Synapse detection
  - VIGRA functions are embedded in *lazyflow* operator objects
  - Operators are connected into workflows (execution graphs)
  - Set of ROIs requested at output
  - *lazyflow* creates set of tasks and executes them in parallel



human input

operator

Visualization of the graph, not visual programming

Conclusions and Outlook

- VIGRA is suitable for very large datasets
- Successfully used in LibreOffice, Hugin Panorama Tools and ilastik
- Users prefer simple syntax over flexibility when possible
- Next steps:
  - Integration of regular (grid-based) and irregular (graph-based) processing
    - Select API (Lemon, boost::graph, own, something else?)
    - Implement graph algorithms without abstraction penalty on grid graphs
  - Parallelization on C++ level
    - Portable framework? (e.g. Posix threads, Intel threading building blocks)
    - Uniform parallelization on multiple levels (loop, thread, GPU, cluster) ? (OpenMP is incompatible with threading, OpenCL ?)
    - Parallel versions of global algorithms (graph cuts, watersheds) – how to achieve satisfactory speed-up
    - Easy-to-use abstractions ? (threads are like spaghetti code)
  - Standardization of generic concepts ?
- Many thanks to all contributors!

Thank You!

- Neuron segmentation in 3D electron microscopy (~2000^3 voxels)
- Uses VIGRA, OpenGM, CGP, CPLEX