The VIGRA Image Analysis Library

Ullrich Köthe, University of Heidelberg

Abstract

VIGRA is a portable and highly flexible library for fundamental image processing and analysis tasks. It uses C++ generic programming techniques to adapt itself to different problems and environments without noticeable abstraction overhead. VIGRA is especially strong in multidimensional data processing and learning-based image analysis. It offers parallelizable Python bindings that are fully integrated with the standard numpy numerical package. Current development efforts concentrate around more advanced parallelization and the unification of regular and irregular (grid- vs. graph-based) processing.

1 Goals

Image analysis is characterized by an abundance of sophisticated algorithms. Implementing and testing these algorithms is often difficult and time-consuming. This leads to a very undesirable situation: Many people put significant effort into the design and development of new algorithms, but use naive solutions for the underlying basic building blocks because better ones would be too costly to implement. For example, several papers describe complex optimization methods for image segmentation, but then use extremely simple image features (e.g. intensity differences between neighboring pixels) for data-dependent energy terms in their objective functions. When these methods are applied, it remains unclear if possible segmentation errors are just caused by the naive features, or by the intrinsic difficulty of the optimization problem. As a consequence, evaluation and comparison with alternative approaches become very hard, since the reasons for observed differences cannot be determined.

Obviously, this dilemma could be solved when reusable versions of important algorithmic building blocks were available. However, research into reusable software components has traditionally focused more on data structures than algorithms. Suitable programming techniques for reusable algorithms have only emerged in the last 15 years or so. The reusability problem in image analysis is further aggravated by the huge amount of data to be processed. It is therefore not acceptable to pay a significant speed penalty for the sake of reusability. This is a major limitation of rapid prototyping platforms such as Matlab: While such platforms contain reusable algorithms for many recurring problems, the techniques used for combining these building blocks into entire applications incur a large abstraction overhead (both in terms of speed and memory) that severely limits the problem sizes which can be reasonably tackled.

VIGRA ("VIsion with GeneRic Algorithms") started in 1997, shortly after the introduction of *generic programming* and *template meta-programming* into the C++ language. It set out to make use of these new techniques for the design of reusable image analysis algorithms that are as efficient as their non-reusable counterparts. Programmers should be able to implement their algorithms on a higher level of abstraction, and let VIGRA's meta-programming functionality take care of code adaptation and optimization for a particular application and environment.

VIGRA takes a conservative attitude to the choice of algorithms it provides: By and large, only algorithms that have stood the test of time are included in the public version. This also means that some fashionable algorithms are missing because their generality has not been demonstrated convincingly in the literature. VIGRA is strong at multidimensional image processing as it always attempts to implement algorithms for arbitrary many dimensions if this is efficiently possible (e.g. convolution and distance transforms). Otherwise, it at least tries to provide versions for 2- and 3-dimensional inputs. All algorithms are subjected to extensive unit and regression testing (almost 1/3 of the VIGRA source code is test code). Quality always takes precedence over quantity.

2 History

VIGRA originates from Ullrich Köthe's PhD project during the late 1990ies, and an extensive description of the early design appears in the thesis [7]. Many of the underlying ideas were also described in a series of papers [5, 10, 6]. VIGRA's original aim was the generalization of the iterator concepts in the C++ standard template library (STL) to two dimensions and the implementation of reusable image processing algorithms on top of theses iterators.

After initial release it turned out that the most frequently requested feature was a module for the transparent import and export of common image file formats. Therefore, the 'impex' module was added in 2002. It sets itself apart from similar libraries (like ImageMagick) because it supports all pixel types (including 'int32', 'float', and 'double', provided the underlying file format can handle them), which is important when image processing is to be used as a measurement and analysis device (and not just as an artistic tool like in PhotoShop or gimp).

A major obstacle in VIGRA's portability was finally resolved in 2003 when Microsoft released its first compiler implementing the required advanced template support. Before that, endless workarounds and compromises were needed to compile VIGRA on Windows. It is hoped that new language features (like C++11) will not again take five years until they make their way to all platforms.

Data structures from discrete geometry such as cell complexes, combinatorial maps, and the GeoMap [11, 13] have played an important role in VIGRA from the very beginning. We were able to design a very convenient and efficient abstract interface for these data structures that was successfully used in generic implementations of many segmentation algorithms, especially hierarchical ones. On several reasons, these algorithms and data structures haven't yet been included into the public VIGRA version.

The next major extension was the introduction of the multidimensional array class in 2003 and the subsequent implementation of multidimensional image processing and analysis algorithms. Multidimensional arrays provide two abstract interfaces – hierarchical iterators and array views – and the latter turned out to be much more convenient for algorithm implementers. Extension of these capabilities has become an ongoing effort since then, including powerful import/export functionality on the basis of the HDF5 standard [1].

The years 2004 and 2005 saw the introduction of dedicated numerical functionality, so that VIGRA no longer depended on external libraries for basic linear algebra and decomposition tasks such as eigenvalue computation. Meanwhile, this part of the library has grown considerably and includes optimization algorithms for least squares, quadratic programming, and sparse decomposition, as well as total variation optimization.

The spline image view classes introduced at around the same time transparently extend the image domain from the integer grid to the real plane and thus provide an indispensable foundation for the subpixel-accurate watershed algorithm [12, 15] and the subpixel GeoMap [13]. Subpixel segmentation results obtained with these methods later formed the basis for a theoretical error analysis which, for the first time, exactly matches the findings of corresponding experimental evaluations [8, 9, 14].

Machine learning started to become an important part of VIGRA in 2009. In particular, the library contains a very fast and flexible implementation of the *random forest* [4] which is quickly gaining popularity in classification and regression due to its low error rate and ease of use.

A big infrastructure effort in 2010 created a completely new build system on the basis of CMake which lead, for the first time, to a truly portable installation process. It also removed the last obstacle for the official release of VIGRA's Python bindings, namely the portable configuration of all required dependencies. The Python bindings itself, which had been around (in a hard-to-install form) since 2004, were completely rewritten to ensure full interoperability with Python's numpy array classes and finally released in 2010 under the name 'vigranumpy'.

Important VIGRA users include LibreOffice and the Hugin Panorama Tools. VIGRA has also been chosen as the underlying number crunching library for the *ilastik* interactive image analysis framework [16]. ilastik is geared towards application experts, especially in the life sciences, who need high quality image analysis algorithms but lack the knowledge for implementing these algorithms themselves. ilastik offers them an intuitive graphical interface where they can interactively provide training data and get immediate feedback on the results, so that the training can be continued until the quality is satisfactory. Due to the large data sets involved here, VIGRA received significant improvements in its on-demand computation and parallelization capabilities. Further improvements in this respect, as well as the implementation of new algorithms, are important ongoing activities of the VIGRA development effort.

3 Contents of the Library

The library consists of two complementary parts: the actual functionality, and the set of abstractions needed to implement this functionality in a generic way. In this section, we only consider the former, the latter will be discussed in section 4. The following list refers to the publicly available functionality.

Images and Multidimensional Arrays
- templated array data structures for arbitrary dimensions and arbitrary pixel/value types
- powerful slicing support similar to Matlab
- input/output of common image file formats (e.g. JPEG, PNG, and TIFF, the latter including 32-bit integer, float and 64-bit double-precision pixel types), input/output of arbitrary dimensional and structured data to HDF5 (including partial loading of huge datasets)

Image Processing
- multidimensional STL-style generic algorithms (e.g. transform, inspect) and functors
- expression templates for efficient expansion of arithmetic and algebraic expressions over arrays
- color space conversions: RGB, sRGB, R'G'B', XYZ, L*a*b*, L*u*v*, Y'PbPr, Y'CbCr, Y'IQ, and Y'UV
- real and complex Fourier transforms in arbitrary dimensions (via FFTW)
- estimation of the camera transfer function and noise distribution

Filters
- convolution in arbitrary dimensions using separable kernels in the spatial domain or arbitrary kernels in the Fourier domain, support for anisotropic data resolution
- filter banks (e.g. Gabor and n-jets) and image pyramids
- image resizing using arbitrary kernels (e.g. splines up to order 5), transparent on-demand interpolation to arbitrary real-valued coordinates using SplineImageView
- non-linear diffusion and total variation filters
- distance transforms and morphological filters in arbitrary dimensions

Image Analysis and Segmentation
- edge and corner detection
- watersheds and region growing (many variants in 2D and 3D)
- region and object statistics using generic accumulators

Machine Learning
- classifiers (random forest, LDA, QDA)
- variable importance, feature selection
- unsupervised decomposition (PCA, PLSA)

Mathematical Tools
- linear algebra (matrix algebra, linear solvers, symmetric and unsymmetric eigenvalues, SVD)
- optimization algorithms (linear and non-negative least squares, ridge regression, L1-constrained least squares, quadratic programming)
- random number generation
- number types (rationals, fixed-point, quaternion) and fixed-size vectors

Prototyping Support
- Python and Matlab bindings

4 Design

The success of any generic library critically depends on the power of the underlying abstractions. A good abstraction must fulfill two conflicting criteria: For the programmer, it should be easy to use in a wide range of contexts, and for the compiler it should be easy to optimize into fast executables. VIGRA is continually in search for better abstractions and has tried a number of different ideas, but the optimal trade-off is very much a moving target: Solutions that were too slow ten years ago are now perfectly OK, because improved compiler and processor technology eliminated the abstraction penalty.

The most important abstractions concern flexible access to the image data. The original design directly generalized the STL concepts to 2D. The resulting `ImageIterators` support navigation functions like `++i.x` to increment the x-coordinate, `i.y += 5` to jump five pixels along the y-coordinate, and `i.x < end.x` to check for the end of iteration. Regions of interest

can be specified by appropriately placing the start and end iterators. Unfortunately, this design could not be extended to multidimensional arrays. Instead, VIGRA opted for hierarchical iterators according to [2]. Here, an iterator always refers to a specific dimension. For example, `++iy` moves the iterator down a *column* (assuming that `iy` refers to the second dimension), and the functions `iy.begin()` and `iy.end()` return a new iterator pair for the current *row*. Starting with an iterator for the outermost dimension, an algorithm can recursively work its way down to the innermost dimension and realize arbitrary navigation patterns in arbitrary dimensions.

An alternative approach, which is easier to comprehend by the user, is the `MultiArrayView` concept which resembles the indexing and slicing functionality pioneered by Matlab. Single points can be accessed via the syntax `view(x,y,z)` or `view[point3d]`, when the view refers to a 3-dimensional array. A view for a region of interest can be created by `view.subarray(start3d, end3d)`, and lower-dimensional slices are obtained by one of several binding functions, e.g. `view.bindOuter(5)` to bind the z-coordinate at the value 5. The index order can be arbitrarily changed without copying the actual data, e.g. `view.transpose()` simply reverses the index order. At the beginning, this design was much slower than the hierarchical iterators because it involves complex address calculations in the inner loop. Meanwhile, compilers are able to optimize this overhead away, and views have become the array interface of choice.

Improvements in compiler technology have also made two additional iterator concepts feasible. `ScanOrderIterator` maps any multidimensional array to a flattened linear sequence according to the current index ordering and the currently selected subarray and slice. Since the scan order of a view generally differs from the memory order of the raw data, this iterator must take care of the appropriate strides. This became as efficient as a hand-crafted loop only recently. A useful variation is the `CoupledIterator` which performs a scan order iteration over up to five arrays and an implicit mesh grid simultaneously.

An important design decision made at the very beginning was the clear separation between algorithms and data structures. Algorithms (usually implemented as free template functions, but sometimes also as classes) can access data only via generic interfaces such as the iterators or array views described above. This has been critical to ensure backwards compatibility in the library's continuing evolution.

Another important part of any generic library is *type inference*. Generic algorithms must be able to determine appropriate types for intermediate variables and end results on the basis of their input types. Since C++ (before C++11) does not support type inference natively, VIGRA comes with a sophisticated set of meta functions ("traits classes") that map input types to output types and reveal other crucial information about the types to be processed. The underlying type conversion rules are defined such that rounding and overflow errors are largely avoided and at the same time memory consumption is kept to a minimum. Important traits provide numeric properties (`NumericTraits`, an extension of `std::numeric_limits`), arithmetic type conversion (`PromoteTraits`), results of norm computations (`NormTraits`), and Python-C++ compatibility rules (`NumpyArrayTraits`), to name just the most important ones.

Powerful algorithms offer a number of user options. Usually, most of these parameters remain at their default values. We need a design where arbitrary subsets of options can easily be set while others are kept alone, and which remains readable at the same time. VIGRA selected a design inspired by Python's keyword arguments: Algorithms are accompanied by option objects, whose constructor sets all options to their default values, and keyword-like functions are used to change these values. For example, watershed seeds can be created by:

```
generateWatershedSeeds(
       srcImageRange(gradient), destImage(seeds),
       SeedOptions().minima().threshold(2.0));
```

when we want to use gradient minima below a value of 2.0.

5 Usage Examples

Basic array syntax is inspired by high-level languages like Matlab and numpy, but without their abstraction penalty:

```
typedef MultiArray<3, double>      Array3;
typedef MultiArrayView<3, double>  View3;
typedef MultiArrayView<2, double>  View2;

Array3 a(Shape3(300,200,100));  // creation
a = 1.0;                        // initialization
a(0,0,0) += a[Shape3(1,2,3)];   // element access
Array3 b = sqrt(a) + 4.0;       // expression templates

View3 s   = a.subarray(Shape3(2,3,4), Shape3(9,8,7));
View2 xy  = a.bindOuter(5);             // slice at z=5
View2 xz  = a.bind<1>(9);               // slice at y=9
View3 yzx = a.transpose(Shape3(1,2,0)); // index order
```

A Gaussian filter with $\sigma = 4$ in arbitrary many dimensions is invoked by

```
gaussianSmoothMultiArray(
     multiArrayRange(a), multiArray(b), 4.0);
```

Options can be set via an option object, e.g. to reduce the filter window size (default is 3σ) to gain speed, to adjust for anisotropic resolution of the data, or to restrict the computation to a subregion:

```
gaussianSmoothMultiArray(
     multiArrayRange(a), multiArray(b), 4.0,
     ConvolutionOptions<3>()
      .filterWindowSize(2.0) // window is 2*σ
      .stepSize(1, 1, 3.2)   // z resolution is lower
      .subarray(Shape3(40,40,10), Shape3(200,60,40)));
```

Region statistics of labeled images can be collected by means of accumulators, and one can freely select the appropriate combination of statistics for each purpose. The actual computation is done using the scan order iterators described earlier:

```
MultiArray<3, UInt32> labels(...);
typedef CoupledIteratorType<3, UInt32>::type Iterator;
AccumlatorChainArray<Iterator::value_type, Select<
    RegionCenter,          // compute region center,
    RegionRadii,           // length of principal axes,
    Coord<Minimum>, Coord<Maximum>   // bounding box
>>  a;     // accumulator array (one for each region)
Iterator start = createCoupledIterator(labels),
         end   = start.getEndIterator();
collectStatistics(start, end, a);
```

Training of a random forest classifier, and class label prediction for new data is accomplished like this:

```
Matrix<float> training_features(n,m), true_labels(n,1),
              features(N,m) /* labels unknown here */;
... // write data into feature and label matrices
RandomForest<float> rf(RFOptions().tree_count(100));
rf.learn(training_features, training_labels);

Matrix<float> class_probabilities(N, rf.class_count());
rf.predictProbabilities(features, class_probabilities);
```

It can be seen that VIGRA puts great emphasis on readability.

6 Lessons Learned and Open Problems

We already mentioned that the best trade-off between power and simplicity of an abstraction on the one hand, and its potential for good optimization and low overhead on the other is difficult to determine and changes with the evolution of compiler technology. Considerations of backward compatibility further complicate these decisions. The time may now be ripe for an interface standardization effort that incorporates the experience of all existing libraries. This would also greatly simplify the interoperation between libraries from different subfields of image analysis.

A related issue is the decision if a certain kind of flexibility should be provided at runtime or at compile time. For example, VIGRA currently fixes the dimensionality of arrays and the loop order for their processing at compile time, whereas other libraries (in particular the prototyping environments numpy and Matlab) leave this to runtime. In general, runtime flexibility leads to more generic code, but compile-time flexibility greatly simplifies optimization. The ultimate solution is probably a sophisticated runtime optimizer, and the speed improvements of the Java Virtual Machine indicate the potential of this solution. But for now, C++ compile time optimization maintains an edge, and the optimal trade-offs between runtime and compile-time flexibility have to be reconsidered regularly.

On the user acceptance side, we found that programmers appreciate flexibility, but also demand that the most common cases are available with a simple syntax. For example, VIGRA implements several variants of the watershed transform by means of a generic region growing algorithm. Programmers must first call a seed creation function, instantiate a functor for the desired cost function, and finally invoke the region growing function, thereby gaining full freedom in the choice of each building block. Nonetheless, these details must be hidden behind an easy-to-use call interface for the most popular alternatives (e.g. the Vincent-Soille algorithm).

Testing has been absolutely crucial for the maintenance and evolution of VIGRA over so many years. The creation of good tests is, however, an open problem: The "correct" result of a complex image analysis algorithm is often unknown, or even undefined when the algorithm involves randomness and uncertainty. A statistical theory of algorithm testing that guides the selection of test cases and judges test results in a probabilistic way would be highly desirable.

One of the most pressing open problems in VIGRA and in image analysis in general is *parallelization*. Unfortunately, many solutions proposed so far are very specific to a particular algorithm, platform, or problem. In some cases (especially for global methods like graph-cuts), parallel algorithms fail to demonstrate satisfactory speed-ups in realistic experiments. A general theory and portable tools for parallelization are largely missing. At present, VIGRA parallelizes mainly on the Python level: Most vigranumpy algorithms can be executed in parallel (e.g. on different tiles of a large dataset), and task allocation and scheduling are performed by our *lazyflow* module (see https://github.com/Ilastik/lazyflow). We are thus able to achieve interactive response times on very large datasets, but in light of the dramatic improvements in parallelization technology that we are currently witnessing, this is certainly not the last word.

Another important open problem is the efficient integration of grid-based and graph-based processing. Two free graph libraries (Lemon and boost::graph) suggest themselves for VIGRA integration, but their APIs differ and are less powerful than the abstractions in our GeoMap framework and related concepts from the discrete geometry community, e.g. the oriented boundary graph [3]. Finding a possibility to unify processing of regular (grid-based) and irregular (graph-based) structures without noticeable speed penalties is a very interesting challenge.

References

[1] HDF5 data storage technologies. http://www.hdfgroup.org/HDF5/, 2012.

[2] M. Austern. Segmented iterators and hierarchical algorithms. In: M. Jazayeri, R. Loos, and D. Musser (eds.): *Generic Programming*, LNCS 1766, pp. 80–90. Springer Berlin / Heidelberg, 2000.

[3] F. Baldacci, A. Braquelaire, and J.-P. Domenger. Oriented boundary graph: A framework to design and implement 3D segmentation algorithms. In: *ICPR 2010*, 2010.

[4] L. Breiman. Random forests. *Machine Learning*, 45(1):5–32, Oct. 2001.

[5] U. Köthe. Design patterns for independent building blocks. In: P. Dyson and J. Coldewey (eds.): *EuroPLoP '98, Proc. 3rd Europ. Conf. Pattern Languages of Programming and Computing*, pp. 143–165, 1998.

[6] U. Köthe. *Reusable Software in Computer Vision*, pp. 103–132. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[7] U. Köthe. *Generische Programmierung für die Bildverarbeitung*. PhD thesis, University of Hamburg, 2000.

[8] U. Köthe. *Reliable Low-Level Image Analysis*. Habilitation Thesis, University of Hamburg, 2008.

[9] U. Köthe. What can we learn from discrete images about the continuous world? In: D. Coeurjolly, I. Sivignon, L. Tougne, and F. Dupont (eds.): *Discrete Geometry for Computer Imagery, Proc. DGCI '08*, Springer LNCS 4992, pp. 4–19, 2008.

[10] U. Köthe and K. Weihe. The stl model in the geometric domain. In: M. Jazayeri, R. Loos, and D. R. Musser (eds.): *Generic Programming*, Springer LNCS 1766, pp. 232–248, 1998.

[11] H. Meine and U. Köthe. The GeoMap: A unified representation for topology and geometry. In: L. Brun and M. Vento (eds.): *Graph-Based Representations in Pattern Recognition, Proc. GbRPR '05*, Springer LNCS 3434, pp. 132–141, 2005.

[12] H. Meine and U. Köthe. Image segmentation with the exact watershed transform. In: *VIIP '05*, 2005.

[13] H. Meine and U. Köthe. A new sub-pixel map for image analysis. In: R. Reulke, U. Eckardt, B. Flach, U. Knauer, and K. Polthier (eds.): *Combinatorial Image Analysis, Proc. IWCIA '06*, Springer LNCS 4040, pp. 116–130, 2006.

[14] H. Meine, U. Köthe, and P. Stelldinger. A topological sampling theorem for robust boundary reconstruction and image segmentation. *Discrete Applied Mathematics*, 157(3):524 – 541, 2009.

[15] H. Meine, P. Stelldinger, and U. Köthe. Pixel approximation errors in common watershed algorithms. In: S. Brlek, C. Reutenauer, and X. Provençal (eds.): *Discrete Geometry for Computer Imagery, Proc. DGCI '09*, Springer LNCS 5810, pp. 193–202, 2009.

[16] C. Sommer, C. Straehle, U. Köthe, and F. Hamprecht. Ilastik: Interactive learning and segmentation toolkit. In: *IEEE International Symposium on Biomedical Imaging*, pp. 230 –233, 30 2011-april 2 2011.