# Shaped Neighborhood based Flood Filled Conditional Iterators

*Release 0.00*

Tim Hauke Heibel and Martin Groher

November 27, 2007

Computer Aided Medical Procedures, Technische Universität München

**Abstract**

ITK's [1] flood filling iterator represents a way to visit pixels/voxels within an image having a specific connectivity. The iterator is initialized at known seed points and starting from these, neighbors that are within the desired connectivity are marked as to be visited and processed by the iterator in the future.

The FloodFilledFunctionConditionalConstIterator builds the foundation for the implementation of the NeighborhoodConnectedImageFilter and is currently fixed to investigating 4-neighborhoods in 2D and 6-neighborhoods in 3D. Since many existing applications use region growing algorithms that perform on full neighborhoods (i.e. 8-connected in 2D and 26-connected in 3D) it is desirable to be able to choose at least between these two standard connectivities.

In this document we describe the extension of the existing iterator to be using ShapedNeighborhoodIterators which has already been proposed in the implementation of the FloodFilledFunctionConditionalConstIterator.

## Contents

Since the changes described in this publication are very basic the remainder will be kept quite short. First we will give a quick overview about the implementation details and then we will provide some results and comparisons of the new iterator with the already existing one. The comparisons will focus on

- topology in 2D and 3D
- speed comparisons.

The topology of the new filter, i.e. the order in which pixels/voxels are processed is here of particular interest since backward compatibility requires the new iterator to process pixels in the exact same order as the existing iterator. This must be the case due to the fact that there may exist implementations which do in some way rely on the order in which pixels/voxels are processed.

## 1 Implementation Details

The existing implementation computes offsets to access a specific pixel in the neighborhood of the currently processed pixel in a low level way. The advantage of this approach is that it is performance wise quite fast though the code is not easy to grasp at the first glance. Furthermore the existing implementation is bound to a fixed 4-neighborhood in 2D and a 6-neighborhood in 3D while some applications might take advantage of fully connected neighborhoods.

It has already been mentioned in the current implementation that the loop used during the neighborhood iteration could be replaced with a `ShapedNeighborhoodIterator` and we simply followed this advice.

Since the processing loop is making extensive use of the actual index of each neighbor, we used the function `ShapedNeighborhoodIterator::ConstIterator::GetNeighborhoodOffset()` and the knowledge about the position of the center pixel to compute the neighbor's index. Due to the fact that we are not using the iterators of the neighborhood to access the underlying pixel values, the `ShapedNeighborhoodIterator` does not need to be shifted over the image itself and thus we are not required to call `ConstNeighborhoodIterator::SetLocation()` which would increase the processing time of the iterator loop dramatically. As a matter of fact we are just using the iterator functionality of the `ShapedNeighborhoodIterator::ConstIterator` but not the one of the `ShapedNeighborhoodIterator` itself.

The proposed `ShapedFloodFilledFunctionConditionalConstIterator` has the same functionality and the same basic interface as the existing `FloodFilledFunctionConditionalConstIterator`. The usage of the `ShapedNeighborhoodIterator` now furthermore allows the user to configure the `ShapedFloodFilledFunctionConditionalConstIterator` such that the flood filling is performed based on a fully connected neighborhood in ND. The class interface has therefore been extended by four functions

- `void SetFullyConnected(const bool fullyConnected)`
- `bool GetFullyConnected() const`
- `void FullyConnectedOn()`
- `void FullyConnectedOff()`

When the member variable `m_FullyConnected` is set to `true` the underlying `ShapedNeighborhoodIterator` is configured to have a fully connected neighborhood (8-connected in 2D and 26-connected in 3D). This configuration is achieved by utilizing the template method `TIterator* setConnectivity( TIterator* it, bool fullyConnected )` provided via the header file itkConnectedComponentAlgorithm.h. Whenever the same member variable is set to `false` the iterator will be put back in its default behavior that is equivalent to the class `FloodFilledFunctionConditionalConstIterator`.

## 2 Results and Testing

The first test we performed aimed at the verification if the new implementation processes contour pixels in the same order as the existing version. This test has been done by creating a synthetic image similar to the one in figure 1 and by executing both iterators on the same image. While incrementing both iterators during each step of the image traversal we check if the iterators are containing the same index.

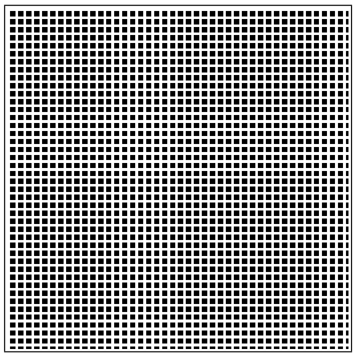

Figure 1: 2D synthetic input

As soon as this is not the case the test will exit while indicating a failure. The test has been implemented currently for 2D case (`IteratorTest2D`) as well as for the 3D case (`IteratorTest3D`) and succeeded in both cases.

For the evaluation of the full connectivity we created an image with an object that could only be extracted as a whole in cases where the iterator investigates a fully connected neighborhood. Again an output image has been created where all pixels visited by the iterator were simply marked with 255. The final test is performed by iterating a second time over all pixels of the input and output image while verifying that both images are identical. The test will exit while yielding a failure if that is not the case and has been implemented in `IteratorTest2D8Connected`.

Finally for the evaluation of the performance compared to the original implementation we created a relatively large image (1280x1280 pixels) and let both iterators process the image one after the other. The processing time has been measured for each iterator separately via the `itk::TimeProbe` class. The process itself has been repeated a hundred times to achieve a more representative measure since the problem with high precision timers is that they do not profile the real performance but rather the performance including processor stalls, which results in measurements that are subject to high fluctuations. Nonetheless the measurements indicate that the new implementation is approximately 20.22% slower than the existing implementation.

For us it is currently not completely clear where this discrepancy is originating from since the `ShapedNeighborhoodIterator` has already all offsets precomputed, and accessing these offsets is just a matter of requesting data from a list. The `ShapedNeighborhoodIterator` itself is, as mentioned beforehand, not recreated during the `ShapedFloodFilledFunctionConditionalConstIterator`'s increment and can thus not be the bottleneck leading to the lack of performance.

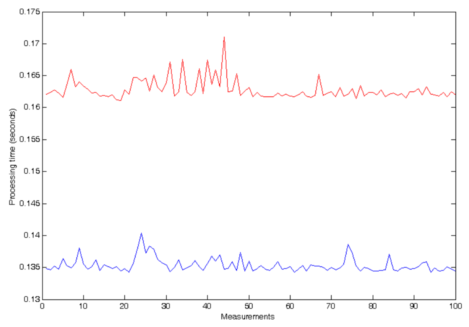

Figure 2: Measured times (Blue 'Low-level Offsets', Red 'ShapedNeighborhoodIterator') for iterating over the test image given in seconds.

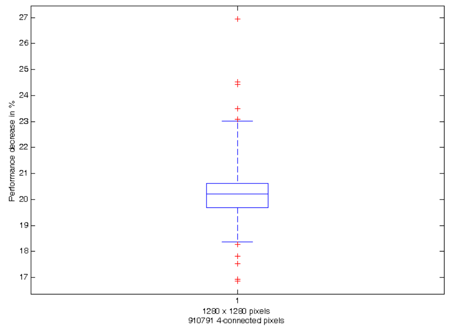

Figure 3: Performance decrease given in percent. The box plot shows a median value of 20.22% as well as quite a few outliers due to the fluctuations of the performance counter.

## 3 Conclusion and Discussion

We have provided a small change for the `FloodFilledFunctionConditionalConstIterator` that allows the user to choose between sparse and full connectivity of the neighborhood. These changes are currently encapsulated in two new classed namely

- `ShapedFloodFilledFunctionConditionalConstIterator`
- `ShapedFloodFilledImageFunctionConditionalConstIterator`.

If it is possible to adapt the new implementation such that it performs equivalently fast as the old implementation we suggest to merge our proposed changes into the existing `FloodFilledFunctionConditionalConstIterator` since it would allow algorithms such as the `NeighborhoodConnectedImageFilter` to perform a region growing on sparsely connected objects.

## References

[1] L. Ibanez, W. Schroeder, L. Ng, and J. Cates. *The ITK Software Guide*. Kitware, Inc. ISBN 1-930934-15-7, http://www.itk.org/ItkSoftwareGuide.pdf, second edition, 2005. (document)