Junior-Prof. Dr. Olaf Ronneberger,
Image Analysis Lab, Institute of Computer Science, Albert-Ludwigs-University Freiburg

Exercises for 3D Image Analysis

Summer term 2015

Exercise 5 (Issue Date: 23.06.2015, Due Date: 07.07.2015)

Dense Elastic Registration

In this exercise you will implement a dense elastic registration. Again, the data will be provided in the hdf5 file format and you can make use of **vivi** or **Fiji** to display the data. With the elastic registration in this exercise we have to minimize a functional of the form

$$E(\mathbf{u}_1, \ldots, \mathbf{u}_N) = \sum_{i=1}^{N} S_i(\mathbf{u}_i) + \lambda \cdot \sum_{i=1}^{N} \sum_{j \in N(i)} \|\mathbf{u}_i - \mathbf{u}_j\|$$

to find the optimal displacement vectors $\mathbf{u}_i$ at each control point $\mathbf{p}_i$, where $i \in \{1, \ldots, N\}$ and N is the number of control points. $N(i)$ is the set of neighbors of point $\mathbf{p}_i$ and defines the edges between the control points in the control point graph. $S_i(\mathbf{u}_i)$ denotes the similarity cost for point i with displacement $\mathbf{u}_i$. The first term of the equation defines the data term (unary potentials). The second term is the smoothness term (pairwise potentials) and defines the smoothness cost of point i with displacement $\mathbf{u}_i$ and neighboring point j with displacement $\mathbf{u}_j$. λ is the weighting factor between the data term and the smoothness term.

To minimize this functional we use **Fast_PD** [1]. In the individual tasks, we compute the dense control point graph, consisting of the set of control points (nodes) and the set of edges, the dense displacement hypotheses (labels) and the unary costs and the pairwise costs.

Note that in this exercise we assign a control point for each pixel in the image. That means, the number of control points N is equal to the number of image pixels, and the corresponding region Ω_i for each control point is just the pixel itself.

To start, create your implementation file and call it "**elastic.cc**".

1) Write a function

```
void computeDenseControlPointGraph(  int nRows, int nCols,
                                     blitz::Array< blitz::TinyVector<int,2>, 1>& nodes,
                                     blitz::Array< blitz::TinyVector<int,2>, 1>& edges)
```

that returns all control point coordinates (the coordinates of all pixels in the image), i.e. the **nodes** and all **edges** of the graph in a one-dimensional `blitz::Array` each. Here we use a 4-connected neighborhood, where all points that have the distance 1 are neighbors. Store the edges as index-pairs of neighboring control points. Use `blitz::TinyVector<int,2>` to store the point coordinates and edge index-pairs in a one-dimensional `blitz::Array` of length nNodes and nEdges respectively. The shape of the control point grid is given as input by nRows and nCols.

2) Write a function that generates a set of dense displacement hypotheses:

```
void computeDenseDisplacementHypotheses(int radius, blitz::Array< blitz::TinyVector<int,2>, 1>& labels)
```

The function returns the displacement hypotheses in the one-dimensional `blitz::Array` **`labels`** of length nLabels, use `blitz::TinyVector<int,2>` for each displacement hypothesis.

Call the two input images **srcIm** and **trgIm**. The estimated deformation field maps the source image onto the target image.

3) Evaluate the **unary cost** at each control point for each displacement hypothesis. Use the sum of squared differences (SSD) as similarity measure.

```
void computeUnaryCostsSSD(        const blitz::Array< blitz::TinyVector<int,2>, 1>& nodes,
                                  const blitz::Array< blitz::TinyVector<int,2>, 1>& labels,
                                  const blitz::Array<float,2>& srcIm,
                                  const blitz::Array<float,2>& trgIm,
                                  blitz::Array< float, 2>& unaryCosts )
```

The output array `unaryCosts` has the shape `nLabels x nNodes.`

4) Evaluate the **pairwise cost** for each pair of displacement hypotheses. Use the l_2 norm of the difference of two vectors to compute the pairwise costs.

```
void computePairwiseCostsL2( const blitz::Array< blitz::TinyVector<int,2>, 1>& labels,
                             blitz::Array< float, 2>& pairwiseCosts )
```

The output array `pairwiseCosts` has the shape `nLabels x nLabels.`

5) To transform the image with the estimated deformation field, you need to implement the function:

```
void warpImage(       blitz::Array<float,2>& warpedBackTrgIm,
                      const blitz::Array<float,2>& trgIm,
                      const blitz::Array<blitz::TinyVector<int,2>,2 >& deformationField)
```

Similar to the previous exercises you perform an inverse transform to warp the target image back into the source image space. Note that you can directly use the forward deformation field. The computed displacements are integer values, so there is no need to interpolate, but you have to be careful at the image borders.

6) Download the code for **Fast_PD** [1] from the course page and read the notes in the file `manual.txt` to understand the interface of the solver.

The constructor of the CV_Fast_PD object has the following form:

```
CV_Fast_PD( int numpoints, int numlabels, Real *lcosts, int numpairs, int *pairs, Real *dist, int
               max_iters, Real *wcosts  )
```

The solver can be used as follows:

```
CV_Fast_PD pd( _numpoints, _numlabels, _lcosts, _numpairs, _pairs, _dist, max_iterations, _wcosts );
pd.run();
```

Note: You can directly use the generated `blitz::Array` structures from above and hand them to the solver interface using the `blitz::Array` dataFirst() pointer.

Here is an example of how to use the solver interface with the `blitz::Array` structures:

```
int max_iterations = 100;
blitz::Array<float, 1> edgeWeights(nEdges);
edgeWeights = 1.0;

float* _unaryCosts = reinterpret_cast<float*>(unaryCosts.dataFirst());
int* _edges  = reinterpret_cast<int*>(edges.dataFirst());
float* _pairwiseCosts   = reinterpret_cast<float*>(pairwiseCosts.dataFirst());
float* _edgeWeights = reinterpret_cast<float*>(edgeWeights.dataFirst());

CV_Fast_PD pd( nNodes, nLabels, _unaryCosts,
               nEdges, _edges, _pairwiseCosts,
               max_iterations, _edgeWeights );
pd.run();

blitz::Array<int, 1> optimalLabels(nNodes);
for( int i = 0; i < nNodes; ++i) {
       optimalLabels(i)=pd._pinfo[i].label;
}

blitz::Array<blitz::TinyVector<int,2>,2 > deformationField(srcIm.shape());

for(int i=0;i < nNodes;++i){
       deformationField(nodes(i)) = labels(optimalLabels(i));
}
```

Don't forget to multiply the pairwise costs with the weighting parameter λ before passing them to the solver. Using `edgeWeights` each edge can be assigned an individual weight. Here, we set these individual weights to 1.

Note: To use the solver in your implementation you have to add the *.cpp files to your compiler string. Your compiler string could look like

```
g++ -Wall -O3 -g elastic.cc  Fast_PD/graph.cpp Fast_PD/LinkedBlockList.cpp Fast_PD/maxflow.cpp -lblitz
-o elastic -lhdf5
```

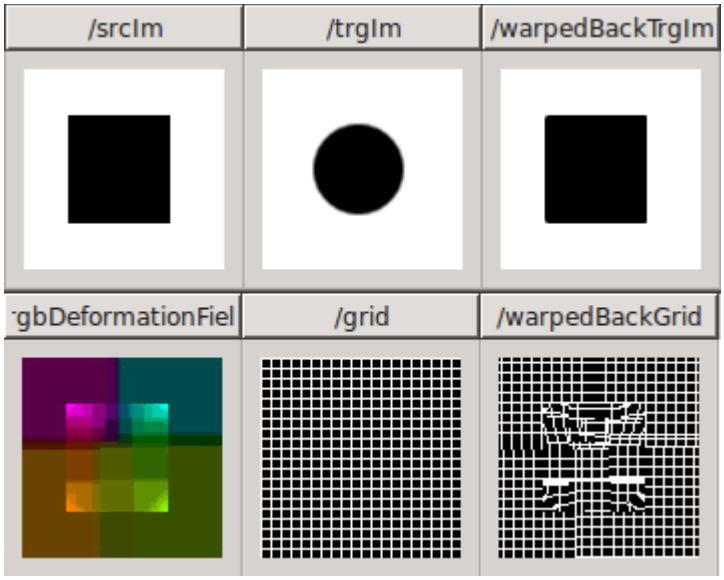

Figure 1: Top row: Source image, target image, warped back target image. Bottom row: Deformation field color coding, target grid and warped back target grid after applying the deformation field.

Figure 2: Color coding of deformation direction. - T. Brox, Optical flow color coding.

7) Write a program elastic.cc, that performs an elastic registration. Use the two sets of toy data (square.h5/circle.h5) and (smile1.h5/smile2.h5). The images are of type `blitz::Array<float,2>` and stored at "/image". To better understand how the deformation looks like, create a grid as in Figure 1 and deform the image and the grid. Additionally, you can visualize the deformation field using the color coding shown in Figure 2. You can use the function flowToImage from the additional material (FlowToImage.hh) to transform the deformation field into an RGB-image. Store your results in a hdf5 file. Experiment with different values for λ.

8) Bonus task: For these datasets the similarity measure SSD was sufficient, but for real world data e.g. recordings from microscope, X-Ray, MRT... it won't be. In those recordings changes in contrast and illumination occur and other similarity measures are more suited. Implement the normalized cross correlation (NCC) as similarity measure:

```
void computeUnaryCostsNCC(    const blitz::Array< blitz::TinyVector<int,2>, 1>& nodes,
       const blitz::Array< blitz::TinyVector<int,2>, 1>& labels,          const blitz::Array<float,2>&
srcIm,                                  const blitz::Array<float,2>& tgrIm,
       blitz::Array< float, 2>& unaryCosts )
```

To compute the NCC you do not use single pixels, but small patches (6x6 pixel) that you compare:

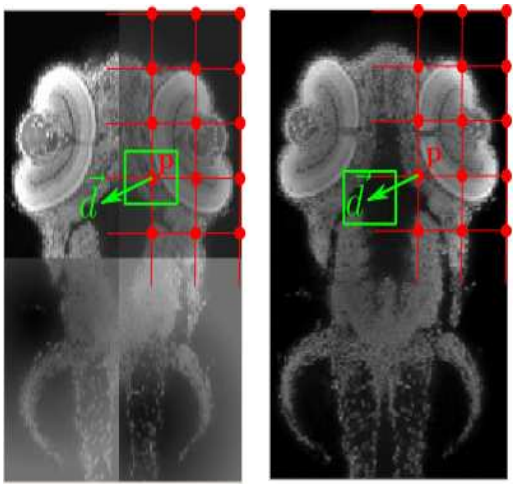

Figure 3: The two zebrafish datasets with different image qualities (gray-level image) and grid of patches (red). The patches for the evaluation of the NCC at point p with displacement d are highlighted in green.

Hint: you have to be careful at the borders, because patches might shoot out of the image. One way to handle this is to do a zero padding around the images.

9) Now apply your dense elastic registration to the zebrafish data (zebra1.h5/zebra2.h5). If you implemented the NCC, compare the results that you obtain with with SSD and NCC (you can also do this on the toy data).

References:

[1]. N. Komodakis and G. Tziritas *"Approximate Labeling via Graph-Cuts Based on Linear Programming"*. IEEE Transactions on Pattern Analysis and Machine Intelligence, 2007.
http://www.csd.uoc.gr/~komod/FastPD/index.html