Exercise: Disparity Map Estimation Using Graph Cuts

1. Problem statement

Given two rectified images:

- Compute a disparity map using a graph cut. See course on graph cuts for details.
- Display the resulting disparity map.
- Compare with the region-growing algorithm: precision, smoothness and computation time.

(You may want to add a slight blur to the result to extra smooth it a bit.)

2. Imagine++ library

The Imagine++ library will let you easily load and operate on images. You will also be able to create a mesh representing the depth map and display it in 3D (except on some Apple computers not supporting openGL). Documentation: see http://imagine.enpc.fr/~monasse/Imagine++/

2.1. Header

The header you need to access library services is as follows:

```
#include <Imagine/Images.h>
using namespace Imagine;
```

2.2. Useful types and functions

- small integer (typically for greyscale intensity) between 0 and 255: `byte`
- image class: `Image<byte>`, `Image<double>`
- image creation: `Image<double> I(w,h)`
- image access: `I(x,y)=0; return I(x+a,y+b);`
- image size: `I.height()`, `I.width()`
- sub-image creation: `I.getSubImage(x,y,w,h)`
- image enlargement by given factor (with interpolation): `enlarge(I,fact)`
- image blurring with Gaussian of given sigma: `blur(I,sigma)`
- representation of a scalar image (e.g., double) by a greyscale (e.g., byte): `bI=grey(dI)`
- reference to file located in source directory: `srcPath("file_in_src_dir.txt")`
- image loading: `load(I,srcPath("face00R.png"))`
- window opening: `openWindow(w,h)`
- image displaying in open window at given offsets: `display(I,x=0,y=0)`
- wait for mouse click in active window: `click()`

3. Graph cut library

A graph cut library is provided in the `maxflow` directory. The library documentation is available as comments in file `maxflow/graph.h`. An example of the use of the library is described in file `maxflow/README.txt` and available in file `exampleGC.cpp`. It can be compiled via `cmake`.

3.1. Header

The header will need to access library services is as follows:

```
#include "maxflow/graph.h"
```

3.2. Useful classes and functions

- graph class (with type of flow/capacities): `Graph<int,int,int>`
- graph creation: `Graph<int,int,int> G(nbNodeMax,nbEdgeMax)`
- graph node creation (excluding preallocated *s* & *t* nodes): `G.add_node(nbNodes)`
- terminal edge (t-link) creation: `G.add_tweights(nodeNum,sWeight,tWeight)`
- n-link creation: `G.add_edge(nodeNum1,nodeNum2,weight12,weight21)`
- max-flow computation: `minE=G.maxflow()`
- type of the source node: `Graph<int,int,int>::SOURCE`
- type of the sink node: `Graph<int,int,int>::SINK`
- type a node after cut: `G.what_segment(nodeNum)`

Note that all n-links are assumed bidirectional. To create mono-directional n-links, use "infinite" weights. For this, use capacities that are much larger that any flow, but do not use `INT_MAX` or `FLT_MAX` as it may overflow after an addition.

4. Provided material

4.1. Code template

A file named `GCDisparity.cpp` is provided. It contains a code skeleton that

- loads two images,
- defines relevant parameters,
- constructs the graph **[this is actually an empty part to be completed ≈ 30 LOC]**,
- computes the maximum flow,
- reads disparities from the minimum cut **[empty part to be completed ≈ 5 LOC]**,
- constructs a 3D mesh from these disparities,
- displays it.

This code can be compiled via `cmake`. However, the result will not make sense until you complete the code as appropriate (see the comments "BEGIN CODE TO BE COMPLETED … END CODE TO BE COMPLETED").

4.2. Images

Two rectified images are provided. (But you can use others! In such case, you have to adjust dmin and dmax) As you will see in the code template, they are clipped a bit (i.e., a small margin is removed) after loading to mostly focus on pixels that are visible in both images.

Here are the values or orders of magnitude of parameters that make sense for the provided images:

- image size: 512x512
- clipping (cropping) margin: 20 to 30 pixels
- NCC neighborhood size: 3 pixel radius, i.e. 7x7 pixel patch
- disparity (not to over-dimension the graph): dmin = 10, dmax=55
- discretization factor to weight the penalty of non-correlating pixels: $w_{cc} = 20$

(as we build a graph with `int` weights, the penalty will then range between 0 and 20),

- penalty for pairwise potentials: $\lambda/w_{cc} = 0.1$, i.e., $\lambda = 0.1 * w_{cc,}$
  (as we build a graph with `int` weights, the penalty must be ≥ 1),

4.3. Optimizations

To make the program faster, a simple scaling mechanism has been defined in the code template. The idea is to down-sample the input images on the fly by a given factor *zoom*. In other words, for an image of size *W*x*H,* instead of looking at pixels (*i,j*) with $0 \leq i < W$ and $0 \leq j < H$, you will only look at pixels (*zoom*i*, *zoom*j*) with $0 \leq i < W/zoom$ and $0 \leq j < H/zoom$.

More precisely, as a patch does not make sense at the image border because of undefined pixels outside the image, you will actually look at pixels (*n+zoom*i*, *n+zoom*j*) with $0 \leq i < (W-2n)/zoom$ and $0 \leq j < (H-2n)/zoom$.

To prevent redundant computations when computing NCC, the code template also precomputes and stores the average image intensity in a patch for each pixel of both images. It is stored into a "pseudo" image whose pixel intensity is that mean value.

4.4. Hints

The graph-cut library works with node number. To clarify the setting, my advice is to create first a formula to associate a unique node number to a given triplet (*x,y,d*) of pixel coordinates and disparity.

The graph-cut library assumes that any edge actually consists of a pair of oriented edges, one in each direction. You will need to weight each of these oriented edges, possibly indicating whether the edge exists or not, or more precisely, whether it can be (easily) cut or not. Think well of the weights you want to assign to each of these oriented edges: 0, "infinity", a constant weight, or a weight that depends on the associated pixel and disparities?

5. Packaging your work

- Program the disparity map estimation,
- Play with the parameters (in particular λ and NCC neighborhood size), possibly change the pictures, and comment what happens then in a short report,
- You can compare with the region-growing method in particular regarding precision, smoothness and computation time, and write your comments in the short report,
- Send **both your code and report** in a single, clean archive (no executable code).