Media ECAD. CLIP tutorial 2.

Image Point and Neighbourhood Operators

John Robinson

2.1 CLIP's point operator functions

Towards the end of Week 1, you implemented a program called bouncethresh.cpp. The code for this is given again here:

```
#include "picture.h"
#include "stdio.h"                                  // To be able to use sprintf below
int main(int argc, char *argv[]) {
        if (argc != 2) {
                cout << "Usage: bouncethresh picname\n";
                return –1;
                }
        picture_of_int inpic(argv[1]);              // If picture is colour, this converts to monochrome
        picture_of_int outpic(inpic.nrows(),inpic.ncols());     // Instantiates a zeroed (black) pictures of
                                                                // the same size as inpic
        outpic.show();
        int threshold = 0;
        int increment = 1;
        while (!outpic.closerequested()) {
                for (int r = 0; r < inpic.nrows(); r++) {
                        for (int c = 0; c < inpic.ncols(); c++) {
                                if (inpic[r][c] < threshold)
                                        outpic[r][c] = 0;
                                else
                                        outpic[r][c] = 255;
                                }
                        }
                threshold += increment;
                if (threshold == 255)
                        increment = -1;
                else if (threshold == 0)
                        increment = 1;
                char comment[256];
                sprintf(comment, "Threshold = %d", threshold);      // Formats an informative string
                                                                    // into the comment buffer…
                outpic.reshow(comment);                             // … which we use to relabel the
                                                                    // window as we show the newly
                                                                    // processed picture.
                }
        }
```

This code features reading and writing from individual pels of a picture_of_int using square brackets as if it were a two-dimensional array. It is always possible to fall back to this kind of programming, but it is usually better to use CLIP member functions that iterate a callback over the picture. "Iterate a callback" sounds intimidating, but don't panic: just follow the examples given here (and check in picture.h for the

functions available and their arguments if you're unsure). The basic idea behind callbacks in CLIP is that you don't want to keep writing pieces of code like:

```
for (int r = 0; r < inpic.nrows(); r++) {
        for (int c = 0; c < inpic.ncols(); c++) {
                if (inpic[r][c] < threshold)
                        outpic[r][c] = 0;
                else
                        outpic[r][c] = 255;
                }
        }
```

because (a) they make your code messy and (b) CLIP can step through every pel faster than you can. Instead you write a function that processes just a single pel, then pass the address of that function to CLIP.

For thresholding, the function on a single pel looks like this:

```
int threshpel(int& outpel, const int& inpel) {
      if (inpel < threshold) outpel = 0;
      else outpel = 255;
      }
```

inpel will come from the input picture and is declared const because you are not going to modify the input. The function tests inpel against the threshold and sets outpel appropriately. To use this function, the variable threshold has to be an external variable, declared above the function as shown in the full program below.

The way you tell CLIP to run this function over the whole picture is with a line like this:

```
outpic.point(inpic, threshpel);
```

where inpic and outpic fulfill the same roles as previously and threshpel is the name of our callback function.

Here then is the full program for implementing bouncethresh using a callback:

```
#include "picture.h"    // The header file for CLIP
#include "stdio.h"      // Required for sprintf
int threshold;
int threshpel(int& outpel, const int& inpel) {
      if (inpel < threshold) outpel = 0;
      else outpel = 255;
      }
int main(int argc, char *argv[]) {
      if (argc != 2) {
            cerr << "Usage: bouncethresh inpic\n";
            return -1;
            }
      picture_of_int inpic(argv[1]);      // Loads pic given in arg 1
      picture_of_int outpic(inpic.nrows(),inpic.ncols());
      outpic.show();
      threshold = 0;
      int increment = 1;
      while(!outpic.closerequested()){
            outpic.point(inpic, threshpel);
            char comment[256];
```

```
            sprintf(comment,"Threshold = %d",threshold);
            outpic.reshow(comment);
            threshold += increment;
            if (threshold >= 255)
                  increment = -1;
            if (threshold <= 0)
                  increment = 1;
            }
      return 0;
      }
```

This code also corrects a bug in last week's bouncethresh program. The threshold displayed as a comment in the window header is now the one actually used for that image. Make sure you understand how this bug arose in the previous version and is fixed now.

Try copying bouncethresh.cpp to btfaster.cpp, editing it to use a threshold callback function, compiling and running. If you compare the speed of the two versions, you should find that the callback version is about 30% faster.

The old bouncethresh and the new btfaster show how to apply an operation to a picture repeatedly, changing the parameters each time. Of course, you usually want either (a) to apply an operation once with a single given parameter, or (b) interactively adjust a parameter until you're satisfied with the result of the operation. The variable thresholding exercise from week 1 (ex 1.2) is an example of a program that does (b). Often though, we just need a single pass of an operation, and there is no need to show() the picture after processing. It just needs to be saved. The following program is a typical example for thresholding. The user will type something like "simplethresh ../pics/goldhill512 outpic 100" and the result is a new file outpic containing the result of thresholding goldhill512 at value 100.

```
// Simple thresholding program
#include "picture.h"    // The header file for CLIP
#include <stdlib.h>     // For prototype of atoi()

int threshold;
int threshpel(int& outpel, const int& inpel) {
      if (inpel < threshold) outpel = 0;
      else outpel = 255;
      }

int main(int argc, char *argv[]) {
      if (argc != 4) {
            cerr << "Usage: simplethresh inpic outpic threshold\n";
            return -1;
            }
      picture_of_int inpic(argv[1]);      // Loads pic given in arg 1
      threshold = atoi(argv[3]);    // Converts threshold string to int
      picture_of_int outpic(inpic, threshpel);
      outpic.write(argv[2]);
      }
```

The above program can be used as a template for any non-interactive point operation program. Depending on the operation you want to do, you may need to change the global variables that get set up at the start – so instead of "int threshold;" you might have "double gammavalue;" for a gamma correction program, for example. But the main thing that changes from program to program is the content of the callback function. Essentially, you can just copy simplethresh.cpp and adapt the callback for gamma correction, clipping,

false colour, or whatever you want to do. However, you should also take care to change the usage error message.

You may have noticed that we declared the callback function as returning int, but didn't actually return a value. If our callback had instead looked like this:

```
int threshpel(int& outpel, const int& inpel) {
      if (inpel < threshold) {
            outpel = 0;
            return 0;
            }
      else {
            outpel = 255;
            return 1;
            }
      }
```

or the more concise:

```
int threshpel(int& outpel, const int& inpel) {
      if (inpel < threshold) {
            outpel = 0;
            return 0;
            }
      outpel = 255;
      return 1;
      }
```

then we could have used the return value from the point() member function to find out how many pels were above the threshold:

```
            int numabovethreshold = outpic.point(inpic, threshpel);
```

In general, the point() member function returns the sum of all the returned values from the callback after it has been applied to every pel in the picture. Often, as in the above program, you may choose to ignore this facility.

There are two other ways to apply a point operator. One is to apply it to a single picture – that is, taking the input and the output from the same picture. Then you need a callback with just a single argument, but you pass it to CLIP via the point() member function in just the same way. The histogram equalization program below gives an example of this usage. See also the program thresh.cpp in the bin/src directory, which is a variant on the above simple thresholding program that uses a single-argument callback and also uses the return value from point(). The other way to apply a point operator is when a picture_of_int (or colour_picture) is being constructed. If you construct with two arguments where the first is a picture, and the second the address of a callback, then the new picture is created by applying the callback to the picture given as an argument. An example of this is given in the first gamma correction program below.

2.2 Gamma correction

Gamma correction is used to compensate for non-linearities in an imaging system. In general, we are seeking a point operator of the following form:

$$s = T(r) = kr^{\gamma}$$

So gamma correction applies a power law operator to every pel of an image. Here's how we do that in a callback:

```
double gammavalue;
double k;
int gammacorrect(int& outpel, const int& inpel) {
      outpel = (int) (k*pow((double) inpel, gammavalue));
      }
```

Just as in the threshold program, the variables that the callback uses are declared above it as external variables. In this case they are gammavalue and k. (We don't use the name gamma for a variable because gamma is the name of a function in the C/C++ maths library. I didn't realize this at first and *did* call my variable gamma; the compiler grumbled, and since I couldn't see any syntax problems I guessed that it was the name that was the problem.) The callback function implements the equation above using the maths library function pow(). pow() takes two arguments, both doubles, and raises the first to the power of the second, returning a double. (You can check what functions like this do by typing "man pow" at the shell prompt.) inpel is an int, so has to be cast to a double before being passed to pow(), then the returned value has to be cast to an int to put in outpel.

Here's a simple program that uses the callback to gamma correct an image.

```
// Simple gamma correction program
#include <math.h>
#include "picture.h"
#include "stdlib.h"
double gammavalue;
double k;
int gammacorrect(int& outpel, const int& inpel) {
      outpel = (int) (k*pow((double) inpel, gammavalue));
      }
int main(int argc, char *argv[]) {
      if (argc != 4) {
            cerr << "Usage: simplegamma inpic outpic gammavalue\n";
            return -1;
            }
      picture_of_int inpic(argv[1]);      // Loads pic given in arg 1
      gammavalue = atof(argv[3]);         // Converts argv[3] string to
                                          // double
      double highest = pow(255.0, gammavalue);  // Find out what highest
                                                // value will be
      k = 255.0/highest;                        // Set k to normalize so
                                                // that 255 gets mapped
                                                // to 255.
      picture_of_int outpic(inpic, gammacorrect);
      outpic.write(argv[2]);
      }
```

This program takes three arguments – an input picture name, an output picture name, and a floating point value for gamma. Enter it, compile it, and try executing it with, for example,
simplegamma ../pics/cameraman256 outpic 0.8
Then view both cameraman256 and outpic together to see the result.

Although simplegamma.cpp is short, it is quite dense. Not only does it use the callback that we have already looked at, it also features constructing a picture by running a callback over another picture in the line

```
      picture_of_int outpic(inpic, gammacorrect);
```

This has just the same effect as if we'd constructed outpic to be the same size as inpic, then done:

```
      outpic.point(inpic, gammacorrect);
```

simplegamma.cpp also illustrates one way to normalize gamma correction. Obviously the equation

$$s = T(r) = kr^{\gamma}$$

maps r = 0 to s = 0, i.e. black to black. Since we normally work with 8-bit images, having values from 0 to 255, it makes sense to map the highest input value, r = 255, to the highest output value, s = 255. The usual way to do this is just as in simplegamma.cpp: Find what the highest value would map to, then work out the scaling factor that would map this to 255 – i.e. multiply by 255 and divide by the highest value.

Try the simplegamma program with a variety of values for gamma.

If you know the gamma of a display device, you can work out the gamma correction required. However, you often want to use gamma correction to get a subjective improvement. That suggests having an interactive program to set the gamma correction. Below I give such a program, based on my solution to last week's exercise 1.2 (the interactive thresholding program). I also use colour_picture objects instead of picture_of_int objects, meaning that the callback will be applied to all three colour channels. This may not be the correct way to gamma correct a colour image – in general, the three channels may need independent correction. However it often suffices for subjective enhancement via a gamma correction.

```
// Gamma correction program
// Want to map y = kx^gamma
#include <math.h>
#include "picture.h"    // The header file for CLIP
#include "stdio.h"      // Required for sprintf
double gammavalue;
double k;
int gammacorrect(int& outpel, const int& inpel) {
      outpel = (int) (k*pow((double) inpel, gammavalue));
      }
int main(int argc, char *argv[]) {
      if (argc != 3) {
            cerr << "Usage: gamma inpic outpic\n";
            return -1;
            }
      colour_picture picture(argv[1]);
      int height = picture.nrows();
      int width = picture.ncols();
      colour_picture gammapic(picture);
      gammapic.show("Gamma = 1");
      gammavalue = 1.0;
      int y = height/2;
      while(gammapic.closerequested() == 0){
            if (!gammapic.pointrightbutton())
                  continue;
            if (gammapic.pointy() == y)
                  continue;
            y = gammapic.pointy();
            gammavalue = (y*3.5)/height;  // Because 3.5 is the highest
                                          // gamma we're likely to want.
            double highest = pow(255.0, gammavalue);
            k = 255.0/highest;
            gammapic.point(picture, gammacorrect);
            char caption[64];
```

```
            sprintf(caption,"Gamma = %f",gammavalue);
            gammapic.reshow(caption);
            }
      gammapic.write(argv[2]);
      return 0;
      }
```

This interactive CLIP program is not much longer than the simplegamma program before. Making a CLIP algorithm *visible* is usually very easy – just insert a few show() and reshow() statements; making it *interactive* is a bit more complicated, but not much. Something to watch out for though is not to confuse the point() member function, which iterates a callback over a picture, with all the other member functions that contain the word "point" and which are to do with the mouse pointer. Giving these member functions such similar names was a design mistake, but I'm afraid you just have to live with it.

2.3 Histogram Equalization

Histogram equalization is a very useful point enhancement method. It is parameter-free – a desirable feature of any algorithm. To implement it, we need to collect a histogram of the frequencies of occurrence of the various pel values in the picture; from this generate a cumulative histogram; normalize the histogram bin values so that 255 maps to 255; then use the result as a lookup table to remap the values. Here is a clip program that does it all:

```
//
// Histogram equalization by cumulative density
//
#include "picture.h"
static int histogram[256];
int map(int& pel) {
      pel = histogram[pel];
      }
int hist(int& pel) {
      histogram[pel]++;
      return 1;
      }
int main(int argc, char *argv[])
      {
      if (argc != 3) {
            cerr << "Usage: histequ inpic outpic\n";
            exit(1);
            }
      picture_of_int in(argv[1]);
      int i;
      for (i = 0; i < 256; i++)
            histogram[i] = 0;       // Initialize to 0
      int numpoints = in.point(hist);// Make histogram
      numpoints /= 256;              // in.point returns total
                                     // number of pels, and we
                                     // want to end up with 1/256
                                     // of this number in each bin.
      for (i = 1; i < 256; i++)
            histogram[i] += histogram[i-1];     // Accumulate
      for (i = 0; i < 256; i++)
            histogram[i] /= numpoints;          // Normalize
      in.point(map);                            // Map values
      in.write(argv[2]);
      }
```

Both of the callbacks used in histequ.cpp work directly on a single picture, so they just take one argument and the call to the point() member function just has one argument.

Try histequ on some representative monochrome pictures.

How would you apply histogram equalization to colour pictures?

A final word on point operations: Several of the simple utility programs in the clip/bin/src directory use point(). Take a look at clip.cpp, for example, which sets the lower and upper pel value limits of a picture.

2.4 Neighbourhood operators

CLIP's provides for neighbourhood or local operators in a similar way to point operators, that is, via callbacks. Here, for example is a 4-point laplacian:

```
#include "picture.h"
int lap(int& pel, const int **buf) {
      pel = 4*buf[0][0]-buf[-1][0]-buf[0][-1]-buf[0][1]-buf[1][0];
      return 0;
      }
int main(int argc, char *argv[]) {
      if (argc != 3) {
            cerr << "Usage: lap4 inpic outpic\n";
            exit(1);
            }
      picture_of_int in(argv[1]);
      picture_of_int out(in,lap); // Generate out from in using lap()
      out.write(argv[2]);
      return 1;
      }
```

The neighbourhood operation used here is done via a constructor, but there is a member function neigh() that works analogously to the point() operator previously discussed. We will see neigh() in a later example.

The neighbourhood operator callback is different from the point operator callback. Instead of a single pel being supplied as input, the user gets access the local neighbourhood in the input picture through what looks like a 2D array. The central pel in this neighbourhood is at buf[0][0], so you could do a point operation simply by using buf[0][0] as input. But you can also access any nearby pel by using coordinates *relative to* the center. I.e. in the statement

```
      pel = 4*buf[0][0]-buf[-1][0]-buf[0][-1]-buf[0][1]-buf[1][0];
```

the pels directly to the left, above, right and below the centre pel are being subtracted from 4 times the center pel.

It is usually illegal to index an array with a negative integer, so what is going on here? When CLIP calls your callback, it provides a pointer to a pointer which you use like a 2D array, but which is really set up so that you access pels in the proximity of the current position. Every time your callback is called (which is the number of pels in the picture) CLIP resets the pointers. Even so, you would think that at the boundaries of the picture, there would be a problem. When the central pel is on the boundary, neighbourhood operators look beyond the boundary of the input picture. CLIP solves this by giving each picture a border, so that you can use quite large neighbourhood operators without getting a memory violation. Normally the entire border is set to 0 (black), but in some cases you may prefer it so that the pel values on the very boundaries of the picture are replicated outwards over the border. The member function dc_pad() provides for this. We will meet this later, but for now it does not matter that the border is set to 0.

As with point operators, this example of a neighbourhood operation can serve as a template for any similar operation. For example, you can convert lap4.cpp into a 3 x 3 averaging filter, just by changing the line

```
      pel = 4*buf[0][0]-buf[-1][0]-buf[0][-1]-buf[0][1]-buf[1][0];
```

to

```
      pel = (buf[-1][-1] + buf[-1][0] +buf[-1][1] + buf[0][-1] +
buf[0][0] + buf[0][1] + buf[1][-1] + buf[1][0] + buf[1][1])/9;
```

or into an arbitrarily-sized local average with:

```
int total = 0;
for (int i = -ysize/2; i <= ysize/2; i++)
      for (int j = -xsize/2; i <= xsize/2; j++)
            total += buf[i][j];
pel = total/(xsize*ysize);
```

This code relies on xsize and ysize being odd. Can you see why? Also, why does it make sense only to use odd sizes for a local averaging filter?

A neighbourhood callback in CLIP can implement any kind of local operation – linear or non-linear. When you implement a linear operation (i.e. a weighted sum of the pels in the neighbourhood), the result is a convolution of the picture with the linear operation. Both lap4.cpp and local averaging are linear operators and therefore can be described as convolutions. Local averaging is convolving with a “box filter” because the shape of the operator (i.e. its impulse response) is like a square box.

When you adapt existing code such as lap4.cpp by altering the callback function, don't forget also to alter the Usage statement at the beginning of the main() function.

2.5 Sobel Edge Detection

Here is a neighbourhood operator program that implements the simple form of Sobel edge detection. Ensure you understand how this works, code it, compile and test. You can use last week's ex1.2 to see the effect of thresholding the output picture.

```
#include "picture.h"
#include <stdlib.h>
// Sobel edge operator. Simple magnitude only.
int sobel(int& pel, const int **buf)
      {
      int horiz = buf[-1][-1] + 2*buf[0][-1] + buf[1][-1];
      horiz -= buf[-1][1] + 2*buf[0][1] + buf[1][1];
      int vert = buf[-1][-1] + 2*buf[-1][0] + buf[-1][1];
      vert -= buf[1][-1] + 2*buf[1][0] + buf[1][1];
      pel = abs(horiz) + abs(vert);
      return 0;
      }
int main(int argc, char *argv[])
      {
      if (argc != 3) {
            cerr << "sobel inpic outpic\n";
            exit(1);
            }
      picture_of_int in(argv[1]);
      picture_of_int out(in,sobel);
```

```
out.write(argv[2]);
}
```

2.6 Aperture Correction

It is possible to adapt lap4.cpp into an aperture correction program. But you can also work on the Unix paradigm of combining special purpose tools. In the binaries directory there are a number of programs that implement simple operations over a whole picture. The programs iadd, isub, imul, and idiv allow you to add, subtract, multiply or divide every pel in a picture by an integer. (Check bin/src/iadd.cpp etc. to see how these are implemented using CLIPs operators for combining pictures.) Similarly the programs padd, psub, pmul and pdiv allow you to add, subtract, multiply and divide one picture by another. Having got your program lap4 working, try the following sequence of commands to test aperture correction:

cp ../pics/goldhill512 inpic	*Because it gets boring always typing ../pics/*
lap4 inpic lappic	*Generate laplacian*
idiv lappic scaledlappic 4	*Scale it*
padd inpic scaledlappic sharppic	*Add the scaled version of the laplacian to original*
clip sharppic outpic 0 255	*Clip the output picture to remove overshoots*
di inpic scaledlappic outpic	*Display input, laplacian and output*

Obviously you can adjust the degree of aperture correction by changing the scaling factor.

2.7 Exercises

Do two of the following exercises including at least one of 1,2,3 and at least one of 4,5,6. Document these in your logbook.

1. A better approximation than lap4.cpp to the isotropic laplacian is obtained by convolving the image with the following filter kernel:

$$\nabla^2 = \begin{bmatrix} -1 & -2 & -1 \\ -2 & 12 & -2 \\ -1 & -2 & -1 \end{bmatrix}$$

   Design, code and demonstrate a CLIP program using the neigh() member function that implements this operator.
2. Implement an n x n median filter program. Demonstrate at a range of filter sizes and on a variety of pictures types.
3. Implement a full Sobel operator program that outputs four pictures: one for the horizontal component, one for the vertical component, one for the gradient magnitude and one for the gradient direction. In the gradient direction picture use integers from 0 to 359 to represent degrees clockwise from north.
4. Implement an interactive aperture correction program where right-button dragging is used to control the weighting factor applied to the laplacian.
5. Implement a program that draws and shows the histogram of an image as an image. Combine this with the interactive gamma correction program so that you can show how the histogram changes as gamma is varied.
6. Investigate what happens when an image is repeatedly convolved with a box filter (i.e. the output from the first convolution is used as the input to the second, and so on). Try to describe the result both theoretically and with clear demonstrations.