Morphology for Color Images via Loewner Order for Matrix Fields

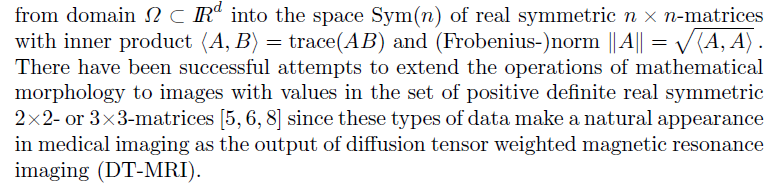
Mathematical morphology is a very successful branch of image processing with a history of more than four decades. Its fundamental operations are dilation and erosion, which are based on the notion of a maximum and a minimum with respect to an order. Many operators constructed from dilation and erosion are available for grey value images,and recently useful analogs of these processes for matrix-valued images have been introduced by taking advantage of the so-called Loewner order. There has been a number of approaches to morphology for vector-valued images, that is, colour images based on various orders, however, each with its merits and shortcomings. In this article we propose an approach to (elementary) morphology for colour images that relies on the existing order based morphology for matrix fields of symmetric 2 × 2-matrices.

Erosion and dilation are the fundamental operations of grey scale morphology relying on the notion of

a minimum and a maximum of real numbers. Since minimum and maximum in turn depend on the presence of an order, it is no surprise that morphology for vector valued i.e. colour images does not always provide satisfactory results. There have been numerous approaches how to extend the mathematical morphology framework to colour or vector-valued images. The main ingredients for

such a framework are ranking schemes and the proper notion of extremal operators such as aximum and minimum. Due to the lack of reasonable complete lattice for vectorial data numerous suggestions for ranking schemes (based on various notions of distances, projections, and real-valued transforms) have been made, for a well structured, comprehensive, in-depth, and still up-to-date survey the reader is referred to [2] and the extensive list of literature cited therein .Depending on the choices made one obtains morphological transforms with specific properties. However, none of these attempts seems to have been accepted unanimously in the image processing community. Somewhat surprisingly, the situation for (symmetric) matrix valued images is not as hopeless as it might seem at first glance.

Here we consider a (symmetric) matrix field F as a mapping:



The goal of this article is to present an approach to morphological operators for colour images by embedding a colour image suitably into a matrix field. Hence the morphology already developed for matrix fields will give rise to morphology for colour images. For the coding of a colour image as a matrix field we will make use of a variant of the HSL-colour space and the Loewner order cone for real symmetric 2 × 2-matrices. This novel concept can be applied to grey value images as well hence it includes scalar (flat) morphology.

Loewner Ordering: Maximal and Minimal Matrices

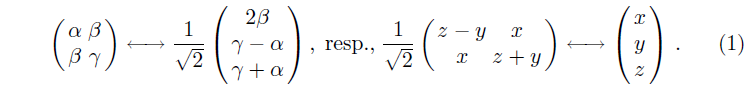
The so-called Loewner order is a natural partial order on Sym(n), defined by means of the cone of positive semidefinite matrices Sym+(n) by:



This partial order is not a lattice order, that is, there is no notion of a unique maximum and minimum with respect to this order [4]. Nevertheless, given any finite set of symmetric matrices A = {A1, . . . ,An}, we will be able to identify suitable maximal, resp., minimal matrices



Since we will consider images with three colour components we may restrict ourselves from now on to the case of 2×2-matrices in Sym(2) which offer already three degrees of freedom. The procedure to find these extremal matrices for a set A is as follows: The cone Sym+(2) can be represented in 3D using the bijection



For A Belonging to Sym(2) the set P(A) = {Z Belong To Sym(2)|A >= Z} denotes the penumbral cone or penumbra for short of the matrix A. It corresponds to a cone with vertex in A and a circular base in the x − y-plane:



Considering the associated penumbras of the matrices in A the search for the maximal matrix A amounts to determine the smallest penumbral cone covering all the penumbras of A tightly, see Fig. 1(b). One realises that the height of a penumbra measured from the x − y-plane is equal to the radius of its base, namely trace(A)/p2. Hence a penumbra is already uniquely determined by the circle constituting its base. This implies that the search for a maximal matrix comes down to find the smallest circle enclosing the base-circles of the matrices in A.



A suitable minimal matrix A is obtained by means of the formula:



with unit matrix I and the parameter *k* = 1/sqrt(2), inspired by the well-known scalar counterpart. For i = 1, . . . , n we have A \_ Ai \_ A with respect to the Loewner order.

Code for colour Morphology:

#include"includes.h"

#include"image.h"

#include"morphological\_function.h"

int main()

{

string file\_src="lenna.ppm";

image img=read\_image(file\_src);

int s\_row,s\_col,s\_x,s\_y;

cout<<"Enter the number of row in the structuring element: ";

cin>>s\_row;

cout<<"Enter the number of column in the structuring element: ";

cin>>s\_col;

cout<<"Enter the x coordinate of origin: ";

cin>>s\_x;

cout<<"Enter the y coordinate of origin: ";

cin>>s\_y;

image e\_img=erosion(img,s\_row,s\_col,s\_x,s\_y);

image d\_img=dilation(img,s\_row,s\_col,s\_x,s\_y);

e\_img.write\_image("eroded\_lenna.ppm");

d\_img.write\_image("dilated\_lenna.ppm");

image o\_img=opening(img,s\_row,s\_col,s\_x,s\_y);

image c\_img=closing(img,s\_row,s\_col,s\_x,s\_y);

o\_img.write\_image("opening\_lenna.ppm");

c\_img.write\_image("closing\_lenna.ppm");

image top\_hat\_filter\_image=top\_hat\_filter(img,s\_row,s\_col,s\_x,s\_y);

top\_hat\_filter\_image.write\_image("top\_hat\_filter\_lenna.ppm");

image s\_image=img-top\_hat\_filter\_image;

s\_image.write\_image("s\_image.ppm");

image black\_top\_hat\_filter\_image=black\_top\_hat\_filter(img,s\_row,s\_col,s\_x,s\_y);

black\_top\_hat\_filter\_image.write\_image("black\_top\_hat\_filter\_lenna.ppm");

image outline\_image=outline(img,s\_row,s\_col,s\_x,s\_y);

outline\_image.write\_image("outline\_lenna.ppm");

image black\_white=black\_whilte(img,s\_row,s\_col);

black\_white.write\_image("black\_white.ppm");

}

Code for operations on Image:

#include"includes.h"

#include"pixel.h"

class image

{

private:

int width,height,range\_max;

public:

vpipi bitmap;

void set\_width(int width)

{

this->width=width;

}

void set\_height(int height)

{

this->height=height;

}

void set\_range\_max(int range\_max)

{

this->range\_max=range\_max;

}

void set\_bitmap(vpipi bitmap)

{

this->bitmap=bitmap;

}

int get\_width()

{

return this->width;

}

int get\_height()

{

return this->height;

}

int get\_range\_max()

{

return this->range\_max;

}

vpipi get\_bitmap()

{

return this->bitmap;

}

image()

{

}

image(int width,int height,int range\_max)

{

this->width=width;

this->height=height;

this->range\_max=range\_max;

bitmap.resize(height,vector<pixel>(width));

}

void print\_image()

{

cout<<width<<" "<<height<<" "<<range\_max<<endl;

for(int i=0; i<height; i++)

{

for(int j=0; j<width; j++)

{

cout<<bitmap[i][j].red<<" "<<bitmap[i][j].green<<" "<<bitmap[i][j].blue<<endl;

}

}

}

void print\_image(int param)

{

cout<<width<<" "<<height<<" "<<range\_max<<endl;

}

void write\_image(string src\_file\_name)

{

int dimx, dimy, range;

dimx=this->width;

dimy=this->height;

range=this->range\_max;

int i, j;

FILE \*fp = fopen(src\_file\_name.c\_str(), "wb"); /\* b - binary mode \*/

(void) fprintf(fp, "P6\n%d %d\n%d\n", dimx, dimy,range);

for (j = 0; j < dimy; ++j)

{

for (i = 0; i < dimx; ++i)

{

static unsigned char color[3];

color[0] = this->bitmap[j][i].red %( range+1); /\* red \*/

color[1] = this->bitmap[j][i].green % (range+1); /\* green \*/

color[2] = this->bitmap[j][i].blue % (range+1); /\* blue \*/

(void) fwrite(color, 1, 3, fp);

}

}

(void) fclose(fp);

}

image duplicate\_image(image img)

{

image dup\_image;

dup\_image.set\_width(img.get\_width());

dup\_image.set\_height(img.get\_height());

dup\_image.set\_range\_max(img.get\_range\_max());

dup\_image.set\_bitmap(img.get\_bitmap());

return dup\_image;

}

image operator-(image& img)

{

image sub\_img;

sub\_img=sub\_img.duplicate\_image(img);

int height=img.get\_height();

int width=img.get\_width();

//cout<<height<<" "<<width<<endl;

for(int i=0;i<height;i++)

{

for(int j=0;j<width;j++)

{

sub\_img.bitmap[i][j]=this->bitmap[i][j]-img.bitmap[i][j];

}

}

return sub\_img;

}

image operator+(image& img)

{

image add\_img;

add\_img=add\_img.duplicate\_image(img);

int height=img.get\_height();

int width=img.get\_width();

//cout<<height<<" "<<width<<endl;

for(int i=0;i<height;i++)

{

for(int j=0;j<width;j++)

{

add\_img.bitmap[i][j]=this->bitmap[i][j]+img.bitmap[i][j];

}

}

return add\_img;

}

};

image read\_image(string src\_file\_name)

{

int buff\_len=256;

FILE \*pf=fopen(src\_file\_name.c\_str(), "rb");

char buf[buff\_len], \*t;

unsigned int w, h, d;

int r;

t = fgets(buf, buff\_len, pf);

do

{

t = fgets(buf, buff\_len, pf);

}

while ( strncmp(buf, "#", 1) == 0 );

r = sscanf(buf, "%u %u", &w, &h);

r = fscanf(pf, "%u", &d);

fseek(pf, 1, SEEK\_CUR);

image img(w,h,d);

for(int i=0; i<h; i++)

{

for(int j=0; j<w; j++)

{

unsigned char arr[3];

fread(arr, sizeof(unsigned char), 3, pf);

pixel px((unsigned int)arr[0],(unsigned int)arr[1],(unsigned int)arr[2]);

img.bitmap[i][j]=px;

}

}

return img;

}

Code for Morphological Functions:

#include"includes.h"

#include"circle\_function.h"

vpp make\_structure(int s\_row,int s\_col,int s\_x,int s\_y)

{

vpp structure(s\_row,vector<pii>(s\_col,mp(0,0)));

for(int i=0;i<s\_row;i++)

{

for(int j=0;j<s\_col;j++)

{

structure[i][j]=mp(j-s\_x,i-s\_y);

}

}

return structure;

}

bool in\_range(int curr\_x,int curr\_y,int h,int w)

{

if(curr\_x>=0 && curr\_x<w && curr\_y>=0 && curr\_y<h)

return true;

else return false;

}

image erosion(image &img,int s\_row,int s\_col,int s\_x,int s\_y)

{

image e\_image;

e\_image=e\_image.duplicate\_image(img);

int height=img.get\_height();

int width=img.get\_width();

vpp structure = make\_structure(s\_row,s\_col,s\_x,s\_y);

for(int i=0;i<height;i++)

{

for(int j=0;j<width;j++)

{

//place the origin of the structure at i,j

vector<pixel> current;

for(int ii=0;ii<s\_row;ii++)

{

for(int jj=0;jj<s\_col;jj++)

{

int x=i+structure[ii][jj].first;

int y=j+structure[ii][jj].second;

if(in\_range(x,y,height,width))

{

current.pb(img.bitmap[x][y]);

}

}

}

e\_image.bitmap[i][j]=find\_min(current);

}

}

return e\_image;

}

image dilation(image &img,int s\_row,int s\_col,int s\_x,int s\_y)

{

image d\_image;

d\_image=d\_image.duplicate\_image(img);

int height=img.get\_height();

int width=img.get\_width();

vpp structure = make\_structure(s\_row,s\_col,s\_x,s\_y);

for(int i=0;i<height;i++)

{

for(int j=0;j<width;j++)

{

vector<pixel> current;

for(int ii=0;ii<s\_row;ii++)

{

for(int jj=0;jj<s\_col;jj++)

{

int x=i+structure[ii][jj].first;

int y=j+structure[ii][jj].second;

if(in\_range(x,y,height,width))

{

current.pb(img.bitmap[x][y]);

}

}

}

d\_image.bitmap[i][j]=find\_max(current);

}

}

return d\_image;

}

image opening(image &src,int s\_row,int s\_col,int s\_x,int s\_y)

{

image eroded=erosion(src,s\_row,s\_col,s\_x,s\_y);

image dilated=dilation(eroded,s\_row,s\_col,s\_x,s\_y);

return dilated;

}

image closing (image &src,int s\_row,int s\_col,int s\_x,int s\_y)

{

image dilated=dilation(src,s\_row,s\_col,s\_x,s\_y);

image eroded=erosion(dilated,s\_row,s\_col,s\_x,s\_y);

return eroded;

}

image top\_hat\_filter(image &src,int s\_row,int s\_col,int s\_x,int s\_y)

{

image o\_img=opening(src,s\_row,s\_col,s\_x,s\_y);

image t\_image=t\_image.duplicate\_image(src);

return t\_image-o\_img;

}

image black\_top\_hat\_filter(image &src,int s\_row,int s\_col,int s\_x,int s\_y)

{

image c\_img=closing(src,s\_row,s\_col,s\_x,s\_y);

image t\_image=t\_image.duplicate\_image(src);

return c\_img-t\_image;

}

image outline(image &src,int s\_row,int s\_col,int s\_x,int s\_y)

{

image dilated=dilation(src,s\_row,s\_col,s\_x,s\_y);

image eroded=erosion(src,s\_row,s\_col,s\_x,s\_y);

image outline;

outline=dilated-eroded;

return outline;

}

image black\_white(image &img,int s\_row,int s\_col)

{

image dup;

dup=black\_white.duplicate\_image();

}