Extended SDM Metrics

Figure D-1 provides a visualization of image texture using SDM's.

Figure D-1. SDM extended metrics

Listing D-1 illustrates the extended SDM metrics from Chapter 3. The code is available online at

<http://www.apress.com/source-code/ComputerVisionMetrics>

Listing D-1. Extended SDM Metrics from Chapter 3

```
/*
**      CREATED 1991 (C) KRIG RESEARCH, SCOTT KRIG - UNPUBLISHED SOFTWARE
**      PORTED TO MAC 2014
**
**              ALL RIGHTS RESERVED
**
**      THIS SOFTWARE MAY BE USED FREELY FOR ACADEMIC AND RESEARCH PURPOSES.
**      REFERENCE THIS BOOK AND PROVIDE THIS NOTICE WHEN USING THE SOFTWARE.
*/
```

```
using namespace std;
```

```
#include <math.h>
#include <stdio.h>
#include <opencv2/opencv.hpp>
#include "/usr/local/include/opencv/cv.h"
#include "/usr/local/include/opencv2/core/core.hpp"
#include "/usr/local/include/opencv2/highgui/highgui.hpp"
#include <iostream>
```

```
using namespace cv;
```

```
#define TINY      0.0000000001
#define F6U      "%6f.3"
#define F6F      "%.6f"
#define F3F      "%.3f"
#define FXF      "%.0f"
#define FALSE 0
#define TRUE 1
```

```
typedef struct area {
    int x;
    int y;
    int dx;
    int dy;
} area_t;
```

```
typedef struct {
    double      t0;
    double      t90;
    double      t135;
    double      t45;
```

```

        double      tave;
    } ctab;

typedef struct {
    double      median;
    double      ave;
    double      adev;
    double      sdev;
    double      svar;
    double      skew;
    double      curt;
    int         min;
    int         max;
    ctab        xcentroid;
    ctab        ycentroid;
    ctab        _asm;
    ctab        low_frequency_coverage;
    ctab        total_coverage;
    ctab        corrected_coverage;
    ctab        total_power;
    ctab        relative_power;
    ctab        locus_length;
    ctab        locus_mean_density;
    ctab        bin_mean_density;
    ctab        containment;
    ctab        linearity;
    ctab        linearity_strength;
    ctab        autocorrelation;
    ctab        covariance;
    ctab        inertia; /* haralick contrast */
    ctab        absolute_value;
    ctab        inverse_difference; /* haralick */
    ctab        entropy; /* haralick */
    ctab        correlation; /* haralick */
} glob_t;

glob_t      gt;

/* FUNCTIONS */

int i_sort(
int      *x,
int      n,
int      parm)
{
    int      k,i,ii;
    int      y,found;

```

```

int      xi;
int      n2, n2p;

x--;
for (k=1; k<n+1; k++) {
    y = x[k];
    for (i=k-1, found = FALSE; i>=0 && !found; ) {
        xi = x[i];
        ii = i+1;
        if (y < xi) {
            x[ii] = xi;
            i--;
        } else {
            found = TRUE;
        }
    }
    x[ii] = y;
}
if (parm == 0) return 0;
n2p = (n2=(n>>1))+1;
return ( n % 2 ? x[n2p] : (x[n2] + x[n2p]) >>1 );
}

```

```

int lmoment(
int      *data,
int      n,
double   *median,
double   *ave,
double   *adev,
double   *sdev,
double   *svar,
double   *skew,
double   *curt)
{
    int      j;
    double   s,p,ep=0.0;

    if (n <= 1) return 0;

    s=0.0;
    for (j=1; j<=n;j++) s += (double)data[j];
    *ave=s/n;
    *adev=(*svar)=(*skew)=(*curt)=0.0;
    for (j=1;j<=n;j++) {
        *adev += abs(s=(double)data[j]-(*ave));
        *svar += (p=s*s);
        *skew += (p *= s);
        *curt += (p *= s);
    }
}

```

```

*adev /=n;
*svar = (*svar - ep*ep / n) / (n-1);
*sdev=sqrt(*svar);
if (*svar) {
    s = (n*(*svar)*(*sdev));
    if (s != 0) *skew /=s;
    else *skew = 0;
    s = (n*(*svar)*(*svar))-3.0;
    if (s != 0) *curt = (*curt) / s;
    else *curt = 0;
} else {
    *skew = *curt = 0.0;
}
*median = 0;
if (n > 20000) return 0;

*median = (double)i_sort(data, n, 1);

return 0;

}

int      mean_sdev(
int      xp,
int      yp,
int      *xdata,
int      *xmean,
double   *xsdev,
double   *ymean,
double   *ysdev)
{
    double   u_x1, a_x1;
    int      mx, my,v,t,x,y,z, offset;
    int      dif[256];

    /* first calculate mean */
    offset = 256 * yp;
    x = y = 0;
    for (z=0; z < 256; x += xdata[offset+z], z++);
    for (z=0; z < 256; y += xdata[xp + (z*256)], z++);

    mx = x / 256.;
    *xmean = (double)mx;
    my = y / 256.;
    *ymean = (double)my;

```

```

/* now calculate standard deviation */
x = y = 0;
z=0;
while (z < 256) {
    v = mx - xdata[offset+z];
    x += v*v;
    v = my - xdata[xp + (z*256)];
    y += v*v;
    z++;
}
*xsdev = x / 256;
*ysdev = y / 256;

return 0;
}

```

```

int lohi(
int      n,
int      *cv,
int      *lo,
int      *hi)
{
    int      x;
    int      lv, hv;

    lv = 0x1fffff;
    hv = 0;
    x=0;
    while (x < n) {
        if (cv[x] < lv) lv = cv[x];
        if (cv[x] > hv) hv = cv[x];
        x++;
    }

    *lo = lv;
    *hi = hv;

    return 0;
}

```

```

int      savegt(
ctab      *ctp,
double    dv1,
double    dv2,

```

```

double dv3,
double dv4)
{

    ctp->t0 = dv1;
    ctp->t90 = dv2;
    ctp->t135 = dv3;
    ctp->t45 = dv4;
    ctp->tave = (dv1 + dv2 + dv3 + dv4) / 4;

    return 0;
}

int      gtput(
char    *prompt,
char    *fs,
char    *ctab,
FILE    *fstream)
{

    char    str[256];
    char    form[256];

    fputs(prompt, fstream);
    sprintf(form, "%s      %s    %s    %s    %s\n", fs, fs, fs, fs, fs);
    sprintf(str, form, ctp->t0, ctp->t90, ctp->t135, ctp->t45, ctp->tave);
    fputs(str, fstream);

    return 0;
}

int      put_txfile(
FILE    *fstream)
{

    char    str[256];

    sprintf(str, "gray value moments: min:%u max:%u mean:%u\n",
            gt.min, gt.max, (int)gt.ave);
    fputs(str, fstream);
    sprintf(str, "moments: adev:%.4f sdev:%.4f svar:%.4f skew:%.6f
            curt:%.6f\n", gt.adev, gt.sdev, gt.svar, gt.skew, gt.curt);
    fputs(str, fstream);
}

```

```

fputs("\\n", fstream);
fputs("-----\\n", fstream);
fputs("          0deg   90deg   135deg   45deg   ave\\n", fstream);
fputs("-----\\n", fstream);
gtput("xcentroid          ", F3F, &gt.xcentroid, fstream);
gtput("ycentroid          ", F3F, &gt.ycentroid, fstream);
gtput("low_frequency_coverage ", F3F, &gt.low_frequency_coverage,
      fstream);
gtput("total_coverage          ", F3F, &gt.total_coverage, fstream);
gtput("corrected_coverage      ", F3F, &gt.corrected_coverage, fstream);
gtput("total_power            ", F3F, &gt.total_power, fstream);
gtput("relative_power          ", F3F, &gt.relative_power, fstream);
gtput("locus_length            ", F3F, &gt.locus_length, fstream);
gtput("locus_mean_density       ", F3F, &gt.locus_mean_density, fstream);
gtput("bin_mean_density         ", F3F, &gt.bin_mean_density, fstream);
gtput("containment             ", F3F, &gt.containment, fstream);
gtput("linearity               ", F3F, &gt.linearity, fstream);
gtput("linearity_strength       ", F3F, &gt.linearity_strength, fstream);

return 0;

}

int    texture(
char   *filename)
{
    char    str[256];
    int     pmx[256], pmy[256];
    int     x,y,z,dx,dy,dz,sz,bpp;
    int     accum, tmin, tmax;
    int     tmin2, tmax2, yc;
    int     *data;
    int     mval0, mval90, mval135, mval45;
    double  median, ave, adev, sdev, svar, skew, curt;
    double  median2, ave2, adev2, sdev2, svar2, skew2, curt2;
    int     *dm0, *dm90, *dm135, *dm45;
    FILE    *fstream;
    int     i0, i90, i135, i45, iave, n;
    int     c0, c90, c135, c45, cave;
    int     p0, p90, p135, p45, pave;
    double  d0, d90, d135, d45, dave;
    double  f0, f90, f135, f45;

```



```

/*****
/* READ THE INPUT IMAGE, EXPECT IT TO BE 8-bit UNSIGNED INT */
/* Mat type conversion is simple in openCV, try it later */

Mat imageIn = cv::imread(filename);
dx = imageIn.rows;
dy = imageIn.cols;
unsigned char *pixels = imageIn.data;

cout << "dx " << dx << " dy " << dy << " elemSize() " <<
        imageIn.elemSize() << endl;
data = (int *)malloc(dx * dy * 4);
if (data == 0)
{
    cout << "malloc error in texture()" << endl;
}
for (y=0; y < dy; y++) {
    for (x=0; x < dx; x++) {
        int pixel = (int)*(imageIn.ptr(x,y));
        if (pixel > 255) { pixel = 255; }
        data[(y * dx) + x] = pixel;
    }
}

/*****
/* PART 1 - get normal types of statistics from pixel data */

lmoment(data, sz, &median, &ave, &adev, &sdev, &svar, &skew, &curt);
lohi(sz, data, &tmin, &tmax);

gt.median = median;
gt.ave = ave;
gt.adev = adev;
gt.sdev = sdev;
gt.svar = svar;
gt.skew = skew;
gt.curt = curt;
gt.min = tmin;
gt.max = tmax;

fstream = fopen("SDMExtended.txt", "w");
if (fstream <= 0) {
    cout << "#cannot create file" << endl;
    return 0;
}

```

```

sprintf(str, "texture for object:      %s\n", filename);
fputs(str, fstream);
sprintf(str, "area:                      %u, %u \n", dx, dy);
fputs(str, fstream);

/*****
/* PART 2 - calculate the 4 spatial dependency matrices */

dm0  = (int *)malloc( 256*256*4 );
dm90 = (int *)malloc( 256*256*4 );
dm135 = (int *)malloc( 256*256*4 );
dm45  = (int *)malloc( 256*256*4 );
if ((dm0==0) || (dm90==0) || (dm135==0) || (dm45==0)) {
    cout << "malloc error in texture2" << endl;
    return 0;
}

x=0;
while (x < 256*256) {
    dm0[x] = dm90[x] = dm135[x] = dm45[x] = 0;
    x++;
}

y=0;
while (y < dy-1) {
    yc = dx * y;

    x=0;
    while (x < dx-1) {

        dm0[ (data[yc + x]&0xff) + ((( data[yc + x + 1] )<<
            8)&0xff00) ]++;
        dm0[ (data[yc + x + 1]&0xff) + ((( data[yc + x] )<<
            8)&0xff00) ]++;
        dm90[ (data[yc + x]&0xff) + ((( data[yc + x + dx] )<<
            8)&0xff00) ]++;
        dm90[ (data[yc + x + dx]&0xff) + ((( data[yc + x] )<<
            8)&0xff00) ]++;
        dm135[ (data[yc + x]&0xff) + ((( data[yc + x + dx + 1]
            )<< 8)&0xff00) ]++;
        dm135[ (data[yc + x + dx + 1]&0xff) + ((( data[yc + x]
            )<< 8)&0xff00) ]++;
        dm45[ (data[yc + x + 1]&0xff) + ((( data[yc + x + dx] )<<
            8)&0xff00) ]++;
        dm45[ (data[yc + x + dx]&0xff) + ((( data[yc + x + 1] )<<
            8)&0xff00) ]++;
    }
}

```

```

        x++;
    }
    y++;
}

/***** CALCULATE TEXTURE METRICS *****/

/* centroid */
pmx[0] = pmx[1] = pmx[2] = pmx[3] = 0;
pmy[0] = pmy[1] = pmy[2] = pmy[3] = 0;
i0 = i90 = i135 = i45 = 0;
y=0;
while (y < 256) {
    x=0;
    while (x < 256) {
        z = x + (256 * y);
        pmx[0] += (x * dm0[z]);
        pmy[0] += (y * dm0[z]); i0 += dm0[z];
        pmx[1] += (x * dm90[z]);
        pmy[1] += (y * dm90[z]); i90 += dm90[z];
        pmx[2] += (x * dm135[z]);
        pmy[2] += (y * dm135[z]); i135 += dm135[z];
        pmx[3] += (x * dm45[z]);
        pmy[3] += (y * dm45[z]); i45 += dm45[z];
        x++;
    }
    y++;
}
pmx[0] = pmx[0] / i0;
pmy[0] = pmy[0] / i0;
pmx[1] = pmx[1] / i90;
pmy[1] = pmy[1] / i90;
pmx[2] = pmx[2] / i135;
pmy[2] = pmy[2] / i135;
pmx[3] = pmx[3] / i45;
pmy[3] = pmy[3] / i45;
x = (pmx[0] + pmx[1] + pmx[2] + pmx[3]) / 4;
y = (pmy[0] + pmy[1] + pmy[2] + pmy[3]) / 4;

gt.xcentroid.t0 = pmx[0];
gt.ycentroid.t0 = pmy[0];
gt.xcentroid.t90 = pmx[1];
gt.ycentroid.t90 = pmy[1];
gt.xcentroid.t135 = pmx[2];
gt.ycentroid.t135 = pmy[2];
gt.xcentroid.t45 = pmx[3];

```

```

gt.ycentroid.t45 = pmy[3];
gt.xcentroid.tave = x;
gt.ycentroid.tave = y;

/* low frequency coverage */
i0 = i90 = i135 = i45 = 0;
c0 = c90 = c135 = c45 = 0;
x=0;
while (x < 256*256) {
    if ((dm0[x] != 0) && (dm0[x] < 3)) i0++;
    if ((dm90[x] != 0) && (dm90[x] < 3)) i90++;
    if ((dm135[x] != 0) && (dm135[x] < 3)) i135++;
    if ((dm45[x] != 0) && (dm45[x] < 3)) i45++;
    if (!dm0[x]) c0++;
    if (!dm90[x]) c90++;
    if (!dm135[x]) c135++;
    if (!dm45[x]) c45++;
    x++;
}
d0 = (double)i0 / 0x10000;
d90 = (double)i90 / 0x10000;
d135 = (double)i135 / 0x10000;
d45 = (double)i45 / 0x10000;

savegt(&gt;.low_frequency_coverage, d0, d90, d135, d45);

d0 = (double)c0 / 0x10000;
d90 = (double)c90 / 0x10000;
d135 = (double)c135 / 0x10000;
d45 = (double)c45 / 0x10000;

savegt(&gt;.total_coverage, d0, d90, d135, d45);

d0 = (c0-i0) / (double)0x10000;
d90 = (c90-i90) / (double)0x10000;
d135 = (c135-i135) / (double)0x10000;
d45 = (c45-i45) / (double)0x10000;

savegt(&gt;.corrected_coverage, d0, d90, d135, d45);

/* power */
i0 = i90 = i135 = i45 = 0;
c0 = c90 = c135 = c45 = 0;
p0 = p90 = p135 = p45 = 0;
y=0;

```

```

while (y < 256) {
    z = y * 256;
    x=0;
    while (x < 256) {
        n = x-y;
        if (n < 0) n = -n;
        if (dm0[x+z] != 0) { i0 += n; c0++; }
        if (dm90[x+z] != 0) { i90 += n; c90++; }
        if (dm135[x+z] != 0) { i135 += n; c135++; }
        if (dm45[x+z] != 0) { i45 += n; c45++; }
        x++;
    }
    y++;
}
d0 = (i0 / 0x10000);
d90 = (i90 / 0x10000);
d135 = (i135 / 0x10000);
d45 = (i45 / 0x10000);

savegt(&gt.total_power, d0, d90, d135, d45);

d0 = (i0 / c0);
d90 = (i90 / c90);
d135 = (i135 / c135);
d45 = (i45 / c45);

savegt(&gt.relative_power, d0, d90, d135, d45);

/* locus density */
d0 = d90 = d135 = d45 = 0.0;
c0 = c90 = c135 = c45 = 0;
p0 = p90 = p135 = p45 = 0;
y=0;
while (y < 256) {
    z = y * 256;
    i0 = i90 = i135 = i45 = 0;
    x=0;
    while (x < 256) {
        n = x-y;
        if (n < 0) n = -n;
        if ((dm0[x+z] != 0) && (n < 7)) { c0++; p0 += dm0[x+z]; }
        if ((dm90[x+z] != 0) && (n < 7)) { c90++; p90 += dm90[x+z]; }
        if ((dm135[x+z] != 0) && (n < 7)) { c135++; p135 +=
            dm135[x+z]; }
        if ((dm45[x+z] != 0) && (n < 7)) { c45++; p45 += dm45[x+z]; }
    }
    y++;
}

```

```

        if ((dm0[x+z] == 0) && (n < 7)) { i0++; }
        if ((dm90[x+z] == 0) && (n < 7)) { i90++; }
        if ((dm135[x+z] == 0) && (n < 7)) { i135++; }
        if ((dm45[x+z] == 0) && (n < 7)) { i45++; }

        x++;
    }
    if (!i0) d0 += 1;
    if (!i90) d90 += 1;
    if (!i135) d135 += 1;
    if (!i45) d45 += 1;
    y++;
}

savegt(&gt;.locus_length, d0, d90, d135, d45);

d0 = (p0/c0);
d90 = (p90/c90);
d135 = (p135/c135);
d45 = (p45/c45);

savegt(&gt;.locus_mean_density, d0, d90, d135, d45);

/* density */
c0 = c90 = c135 = c45 = 0;
p0 = p90 = p135 = p45 = 0;
x=0;
while ( x < 256*256) {
    if (dm0[x] != 0) { c0 += dm0[x]; p0++; }
    if (dm90[x] != 0) { c90 += dm90[x]; p90++; }
    if (dm135[x] != 0) { c135 += dm135[x]; p135++; }
    if (dm45[x] != 0) { c45 += dm45[x]; p45++; }
    x++;
}
d0 = c0 / p0;
d90 = c90 / p90;
d135 = c135 / p135;
d45 = c45 / p45;

savegt(&gt;.bin_mean_density, d0, d90, d135, d45);

/* containment */
i0 = i90 = i135 = i45 = 0;
x=0;
while (x < 256) {
    if (dm0[x]) i0++; if (dm0[256*256 - x - 1]) i0++;
    if (dm90[x]) i90++; if (dm90[256*256 - x - 1]) i90++;

```

```

    if (dm135[x]) i135++; if (dm135[256*256 - x - 1]) i135++;
    if (dm45[x]) i45++; if (dm45[256*256 - x - 1]) i45++;

    if (dm0[x*256]) i0++; if (dm0[(x*256)+255]) i0++;
    if (dm90[x*256]) i90++; if (dm90[(x*256)+255]) i90++;
    if (dm135[x*256]) i135++; if (dm135[(x*256)+255]) i135++;
    if (dm45[x*256]) i45++; if (dm45[(x*256)+255]) i45++;

    x++;
}

d0 = 1.0 - ((double)i0 / 1024.0);
d90 = 1.0 - ((double)i90 / 1024.0);
d135 = 1.0 - ((double)i135 / 1024.0);
d45 = 1.0 - ((double)i45 / 1024.0);

savegt(&gt.containment, d0, d90, d135, d45);

/* linearity */
i0 = i90 = i135 = i45 = 0;
c0 = c90 = c135 = c45 = 0;
y=0;
while (y < 256) {
    z = y * 256;
    if (dm0[z + y] > 1) { i0++; c0 += dm0[z+y]; }
    if (dm90[z + y] > 1) { i90++; c90 += dm90[z+y]; }
    if (dm135[z + y] > 1) { i135++; c135 += dm135[z+y]; }
    if (dm45[z + y] > 1) { i45++; c45 += dm45[z+y]; }
    y++;
}
d0 = (double)i0 / 256.;
d90 = (double)i90 / 256.;
d135 = (double)i135 / 256.;
d45 = (double)i45 / 256.;

savegt(&gt.linearity, d0, d90, d135, d45);

/* linearity strength */
d0 = (c0/(i0+.00001)) / 256.;
d90 = (c90/(i90+.00001)) / 256.;
d135 = (c135/(i135+.00001)) / 256.;
d45 = (c45/(i45+.00001)) / 256.;

savegt(&gt.linearity_strength, d0, d90, d135, d45);

/* WRITE ALL STATISTICS IN gt. STRUCTURE TO OUTPUT FILE */
put_txfile(fstream);

```

```

/* clip to max value 255 */
mval0 = mval90 = mval135 = mval45 = 0;
x=0;
while (x < 256*256) {
    if (dm0[x] > 255) dm0[x] = 255;
    if (dm90[x] > 255) dm90[x] = 255;
    if (dm135[x] > 255) dm135[x] = 255;
    if (dm45[x] > 255) dm45[x] = 255;
    x++;
}

/*****/
/* Convert data to unsigned char to write into png */

unsigned char *dm0b = (unsigned char *)malloc( 256*256);
unsigned char *dm90b = (unsigned char *)malloc( 256*256);
unsigned char *dm135b = (unsigned char *)malloc( 256*256);
unsigned char *dm45b = (unsigned char *)malloc( 256*256);
if ((dm0b==0) || (dm90b==0) || (dm135b==0) || (dm45b==0)) {
    cout << "malloc error in texture3" << endl;
    return 0;
}

x=0;
while (x < 256*256) {
    dm0b[x] = (unsigned char) (dm0[x] & 0xff);
    dm90b[x] = (unsigned char) (dm90[x] & 0xff);
    dm135b[x] = (unsigned char) (dm135[x] & 0xff);
    dm45b[x] = (unsigned char) (dm45[x] & 0xff);
    x++;
}

/*
* write output to 4 quadrants:      0=0, 1=90, 2=135, 3=145
*/

char outfile[256];

sprintf(outfile, "%s_SDMQadrant_0deg_8UC1.png", filename);
Mat SDMQuadrant0(256, 256, CV_8UC1, dm0b);
imwrite(outfile, SDMQuadrant0);
sprintf(outfile, "%s_SDMQadrant_90deg_8UC1.png", filename);
Mat SDMQuadrant90(256, 256, CV_8UC1, dm90b);
imwrite(outfile, SDMQuadrant90);
sprintf(outfile, "%s_SDMQadrant_135deg_8UC1.png", filename);
Mat SDMQuadrant135(256, 256, CV_8UC1, dm135b);
imwrite(outfile, SDMQuadrant135);

```



```

sprintf(outfile, "%s_SDMQuadrant_45deg_8UC1.png", filename);
Mat SDMQuadrant45(256, 256, CV_8UC1, dm45b);
imwrite(outfile, SDMQuadrant45);

free(dm0);
free(dm90);
free(dm135);
free(dm45);
free(data);
free(dm0b);
free(dm90b);
free(dm135b);
free(dm45b);
fclose(fstream);

return 0;

}

int main (int argc, char **argv)
{
    cout << "8-bit unsigned image expected as input" << endl;
    texture (argv[1]);
    return 0;
}

```