kf-extras

Claude Heiland-Allen

2014–2019

Contents

1 .gitignore

```
kfb-curvature
kfb-de-histogram
kfb-expmap
kfb-histogram
kfb-pseudo-de
kfb-rainbow
kfb-resize
kfb-statistics
kfb-stretch
kfb-to-exr
kfb-to-mmit
*.o
```

2 kfb.c

```
#include <assert.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
```

```
#include "kfb.h"

typedef struct colour { unsigned char r, g, b; } colour;

struct kfb {
  size_t width;
  size_t height;
  int *counts;
  int iterdiv;
  int parts;
  colour *keys;
  int maxiter;
  float *trans;
  int hasde;
  float *de;
};

extern void kfb_free(kfb *k) {
  if (k) {
    if (k->trans) { free(k->trans); k->trans = 0; }
    if (k->keys) { free(k->keys); k->keys = 0; }
    if (k->counts) { free(k->counts); k->counts = 0; }
    if (k->de) { free(k->de); k->de = 0; }
    free(k);
  }
}

extern kfb *kfb_alloc(const kfb *proto, size_t width, size_t height) {
  assert(proto);
  assert(proto->keys);
  kfb *k = (kfb *) calloc(1, sizeof(struct kfb));
  if (! k) { return 0; }
  k->width = width;
  k->height = height;
  k->counts = (int *) calloc(1, width * height * sizeof(int));
  if (! k->counts) { goto fail; }
  k->trans = (float *) calloc(1, width * height * sizeof(float));
  if (! k->trans) { goto fail; }
  k->iterdiv = proto->iterdiv;
  k->parts = proto->parts;
  k->keys = (struct colour *) malloc(k->parts * sizeof(struct colour));
  if (! k->keys) { goto fail; }
  memcpy(k->keys, proto->keys, k->parts * sizeof(struct colour));
  k->maxiter = proto->maxiter;
  k->de = (float *) calloc(1, width * height * sizeof(float));
  return k;
fail:
  kfb_free(k);
  return 0;
}

extern kfb *kfb_load(FILE *in) {
  // FIXME check little-endian host, or convert
  assert(sizeof(int) == 4);
  assert(sizeof(float) == 4);
  assert(in);
```

```
  int mk = getc(in);
  int mf = getc(in);
  int mb = getc(in);
  if (mk != 'K' || mf != 'F' || mb != 'B') { return 0; }
  kfb *k = (kfb *) calloc(1, sizeof(struct kfb));
  if (! k) { return 0; }
  int w = 0;
  int h = 0;
  if (1 != fread(&w, sizeof(int), 1, in)) { goto fail; }
  k->width = w;
  if (1 != fread(&h, sizeof(int), 1, in)) { goto fail; }
  k->height = h;
  if (! (k->counts = (int *) malloc(k->width * k->height * sizeof(int)))) { goto fail; }
  if (1 != fread(k->counts, k->width * k->height * sizeof(int), 1, in)) { goto fail; }
  if (1 != fread(&k->iterdiv, sizeof(int), 1, in)) { goto fail; }
  if (1 != fread(&k->parts, sizeof(int), 1, in)) { goto fail; }
  if (! (k->keys = (struct colour *) malloc(k->parts * sizeof(struct colour)))) { goto fail; }
  if (1 != fread(k->keys, k->parts * sizeof(struct colour), 1, in)) { goto fail; }
  if (1 != fread(&k->maxiter, sizeof(int), 1, in)) { goto fail; }
  if (! (k->trans = (float *) malloc(k->width * k->height * sizeof(float)))) { goto fail; }
  if (1 != fread(k->trans, k->width * k->height * sizeof(float), 1, in)) { goto fail; }
  if (! (k->de = (float *) malloc(k->width * k->height * sizeof(float)))) { goto fail; }
  if (1 != fread(k->de, k->width * k->height * sizeof(float), 1, in)) { k->hasde = 0; } else { k->hasde = 1; }
  return k;
fail:
  kfb_free(k);
  return 0;
}

extern int kfb_save(const kfb *kfb, FILE *out) {
  // FIXME check little-endian host, or convert
  assert(sizeof(int) == 4);
  assert(sizeof(float) == 4);
  assert(out);
  assert(kfb);
  assert(kfb->counts);
  assert(kfb->keys);
  assert(kfb->trans);
  putc('K', out);
  putc('F', out);
  putc('B', out);
  int w = kfb->width;
  int h = kfb->height;
  fwrite(&w, sizeof(int), 1, out);
  fwrite(&h, sizeof(int), 1, out);
  fwrite(kfb->counts, kfb->width * kfb->height * sizeof(int), 1, out);
  fwrite(&kfb->iterdiv, sizeof(int), 1, out);
  fwrite(&kfb->parts, sizeof(int), 1, out);
  fwrite(kfb->keys, kfb->parts * sizeof(colour), 1, out);
```

```
  fwrite(&kfb->maxiter, sizeof(int), 1, out);
  fwrite(kfb->trans, kfb->width * kfb->height * sizeof(float), 1, out);
  return kfb->hasde ? 1 == fwrite(kfb->de, kfb->width * kfb->height * sizeof(↙
      ↳ float), 1, out) : 1;
}

extern size_t kfb_width(const kfb *kfb) {
  assert(kfb);
  return kfb->width;
}

extern size_t kfb_height(const kfb *kfb) {
  assert(kfb);
  return kfb->height;
}

extern double kfb_maxiter(const kfb *kfb) {
  assert(kfb);
  return kfb->maxiter;
}

extern int kfb_get_hasde(const kfb *kfb) {
  assert(kfb);
  return kfb->hasde;
}

extern double kfb_get(const kfb *kfb, size_t i, size_t j) {
  assert(kfb);
  assert(i < kfb->width);
  assert(j < kfb->height);
  size_t k = kfb->height * i + j;
  int c = kfb->counts[k];
  if (c == kfb->maxiter) {
    return kfb->maxiter;
  } else {
    double t = kfb->trans[k];
    if (t > 1.0) {
      return -c; // glitch
    } else {
      return c + (1.0 - t);
    }
  }
}

extern double kfb_get_de(const kfb *kfb, size_t i, size_t j) {
  assert(kfb);
  assert(i < kfb->width);
  assert(j < kfb->height);
  size_t k = kfb->height * i + j;
  int c = kfb->counts[k];
  if (c == kfb->maxiter) {
    return -1;
  } else {
    return kfb->de[k];
  }
}
```

```
static int min(int a, int b) {
  return a < b ? a : b;
}

static int max(int a, int b) {
  return a > b ? a : b;
}

extern double kfb_get_nearest(const kfb *kfb, double i, double j) {
  assert(kfb);
  int i0 = round(i);
  i0 = min(max(i0, 0), kfb->width - 1);
  int j0 = round(j);
  j0 = min(max(j0, 0), kfb->height - 1);
  return kfb_get(kfb, i0, j0);
}

static double linear(double t, double a, double b) {
  return (1 - t) * a + t * b;
}

extern double kfb_get_linear(const kfb *kfb, double i, double j) {
  assert(kfb);
  int i0 = floor(i);
  double x = i - i0;
  int i1 = i0 + 1;
  i0 = min(max(i0, 0), kfb->width - 1);
  i1 = min(max(i1, 0), kfb->width - 1);
  int j0 = floor(j);
  double y = j - j0;
  int j1 = j0 + 1;
  j0 = min(max(j0, 0), kfb->height - 1);
  j1 = min(max(j1, 0), kfb->height - 1);
  double n00 = kfb_get(kfb, i0, j0);
  double n01 = kfb_get(kfb, i0, j1);
  double n10 = kfb_get(kfb, i1, j0);
  double n11 = kfb_get(kfb, i1, j1);
  double n0 = linear(y, n00, n01);
  double n1 = linear(y, n10, n11);
  double n = linear(x, n0, n1);
  return n;
}

static double cubic(double t, double a, double b, double c, double d) {
  const double m[16] = { 0.0,2.0,0.0,0.0, -1.0,0.0,1.0,0.0, 2.0,-5.0,4.0,-1.0, ↙
      ↳ -1.0,3.0,-3.0,1.0 };
  double f[4] = { 1.0, t, t*t, t*t*t };
  double z[4] = { a, b, c, d };
  double r = 0.0;
  for (int i = 0; i < 4; ++i) {
    double s = 0.0;
    for (int j = 0; j < 4; ++j) {
      s += f[i] * m[i * 4 + j];
    }
    r += s * z[i];
  }
  return 0.5 * r;
```

```
  }

extern double kfb_get_cubic(const kfb *kfb, double i, double j) {
  assert(kfb);
  int i1 = floor(i);
  double x = i - i1;
  int i0 = i1 - 1;
  int i2 = i1 + 1;
  int i3 = i1 + 2;
  i0 = min(max(i0, 0), kfb->width - 1);
  i1 = min(max(i1, 0), kfb->width - 1);
  i2 = min(max(i2, 0), kfb->width - 1);
  i3 = min(max(i3, 0), kfb->width - 1);
  int j1 = floor(j);
  double y = j - j1;
  int j0 = j1 - 1;
  int j2 = j1 + 1;
  int j3 = j1 + 2;
  j0 = min(max(j0, 0), kfb->height - 1);
  j1 = min(max(j1, 0), kfb->height - 1);
  j2 = min(max(j2, 0), kfb->height - 1);
  j3 = min(max(j3, 0), kfb->height - 1);
  double n00 = kfb_get(kfb, i0, j0);
  double n01 = kfb_get(kfb, i0, j1);
  double n02 = kfb_get(kfb, i0, j2);
  double n03 = kfb_get(kfb, i0, j3);
  double n10 = kfb_get(kfb, i1, j0);
  double n11 = kfb_get(kfb, i1, j1);
  double n12 = kfb_get(kfb, i1, j2);
  double n13 = kfb_get(kfb, i1, j3);
  double n20 = kfb_get(kfb, i2, j0);
  double n21 = kfb_get(kfb, i2, j1);
  double n22 = kfb_get(kfb, i2, j2);
  double n23 = kfb_get(kfb, i2, j3);
  double n30 = kfb_get(kfb, i3, j0);
  double n31 = kfb_get(kfb, i3, j1);
  double n32 = kfb_get(kfb, i3, j2);
  double n33 = kfb_get(kfb, i3, j3);
  double n0 = cubic(y, n00, n01, n02, n03);
  double n1 = cubic(y, n10, n11, n12, n13);
  double n2 = cubic(y, n20, n21, n22, n23);
  double n3 = cubic(y, n30, n31, n32, n33);
  double n = cubic(x, n0, n1, n2, n3);
  return n;
}

extern void kfb_set(kfb *kfb, size_t i, size_t j, double n) {
  assert(kfb);
  assert(i < kfb->width);
  assert(j < kfb->height);
  int c;
  float f;
  if (n >= kfb->maxiter) {
    c = kfb->maxiter;
    f = 1.0f;
  } else {
    c = floor(n);
```

```
      f = 1.0 - (n - c);
    }
    size_t k = kfb->height * i + j;
    kfb->counts[k] = c;
    kfb->trans[k] = f;
}

extern void kfb_set_hasde(kfb *kfb, int hasde) {
  assert(kfb);
  kfb->hasde = hasde;
}

extern void kfb_set_de(kfb *kfb, size_t i, size_t j, double de) {
  assert(kfb);
  assert(i < kfb->width);
  assert(j < kfb->height);
  int k = kfb->height * i + j;
  kfb->de[k] = de;
}


extern int kfb_get_raw_count(const kfb *kfb, size_t i, size_t j) {
  assert(kfb);
  assert(i < kfb->width);
  assert(j < kfb->height);
  size_t k = kfb->height * i + j;
  int c = kfb->counts[k];
  return c;
}

extern float kfb_get_raw_trans(const kfb *kfb, size_t i, size_t j) {
  assert(kfb);
  assert(i < kfb->width);
  assert(j < kfb->height);
  size_t k = kfb->height * i + j;
  float t = kfb->trans[k];
  return t;
}

extern float kfb_get_raw_de(const kfb *kfb, size_t i, size_t j) {
  assert(kfb);
  assert(i < kfb->width);
  assert(j < kfb->height);
  size_t k = kfb->height * i + j;
  float d = kfb->de[k];
  return d;
}
```

3 kfb-curvature.cc

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

#include "kfb.h"
#include "pgm.h"
```

```
typedef double float1;

struct float2
{
  double x, y;
};

float2 float2_(double x, double y)
{
  float2 m;
  m.x = x;
  m.y = y;
  return m;
}

float2 operator+(float2 u, float2 v)
{
  float2 w;
  w.x = u.x + v.x;
  w.y = u.y + v.y;
  return w;
}

float2 operator/(float2 u, double v)
{
  float2 w;
  w.x = u.x / v;
  w.y = u.y / v;
  return w;
}

double dot(float2 u, float2 v)
{
  return u.x * v.x + u.y * v.y;
}

struct float3
{
  double x, y, z;
};

float3 float3_(float1 x, float1 y, float1 z)
{
  float3 m;
  m.x = x;
  m.y = y;
  m.z = z;
  return m;
}

float3 operator/(float3 u, double v)
{
  float3 w;
  w.x = u.x / v;
  w.y = u.y / v;
  w.z = u.z / v;
  return w;
```

```
65  }

    float3 operator-(float3 v)
    {
      float3 w;
70    w.x = - v.x;
      w.y = - v.y;
      w.z = - v.z;
      return w;
    }
75
    float3 operator-(float3 u, float3 v)
    {
      float3 w;
      w.x = u.x - v.x;
80    w.y = u.y - v.y;
      w.z = u.z - v.z;
      return w;
    }

85  float3 operator+(float3 u, float3 v)
    {
      float3 w;
      w.x = u.x + v.x;
      w.y = u.y + v.y;
90    w.z = u.z + v.z;
      return w;
    }

    double dot(float3 u, float3 v)
95  {
      return u.x * v.x + u.y * v.y + u.z * v.z;
    }

    double length(float3 v)
100 {
      return sqrt(dot(v, v));
    }

    float3 normalize(float3 v)
105 {
      return v / length(v);
    }

    float3 cross(float3 u, float3 v)
110 {
      float3 w;
      w.x = u.y * v.z - u.z * v.y;
      w.y = u.z * v.x - u.x * v.z;
      w.z = u.x * v.y - u.y * v.x;
115   return w;
    }

    struct float3x3
    {
120   float3 x, y, z; // columns
    };
```

```
float3x3 float3x3_(float3 x, float3 y, float3 z)
{
  float3x3 m;
  m.x = x;
  m.y = y;
  m.z = z;
  return m;
}

float3 mul(float3x3 m, float3 v)
{
  float3 w;
  w.x = m.x.x * v.x + m.y.x * v.y + m.z.x * v.z;
  w.y = m.x.y * v.x + m.y.y * v.y + m.z.y * v.z;
  w.z = m.x.z * v.x + m.y.z * v.y + m.z.z * v.z;
  return w;
}

struct float4
{
  double x, y, z, w;
};

float4 float4_(double x, double y, double z, double w)
{
  float4 o;
  o.x = x;
  o.y = y;
  o.z = z;
  o.w = w;
  return o;
}

double dot(float4 u, float4 v)
{
  return u.x * v.x + u.y * v.y + u.z * v.z + u.w * v.w;
}

float4 operator*(double u, float4 v)
{
  float4 o;
  o.x = u * v.x;
  o.y = u * v.y;
  o.z = u * v.z;
  o.w = u * v.w;
  return o;
}

typedef double float4x4[4][4];

float4 mul(float4 v, float4x4 m)
{
  float4 w;
  w.x = v.x * m[0][0] + v.y * m[0][1] + v.z * m[0][2] + v.w * m[0][3];
  w.y = v.x * m[1][0] + v.y * m[1][1] + v.z * m[1][2] + v.w * m[1][3];
  w.z = v.x * m[2][0] + v.y * m[2][1] + v.z * m[2][2] + v.w * m[2][3];
```

```
    w.w = v.x * m[3][0] + v.y * m[3][1] + v.z * m[3][2] + v.w * m[3][3];
    return w;
}

#define TRIANGLE_VERTICES 3
#define LOCAL_X 0
#define LOCAL_Y 1
#define LOCAL_Z 2

// http://graphics.zcu.cz/sscurvature.html
                        float2 CalcCurvature_ShapeOperator_2edge(float3 a, float3 b, float3 c, float3 an, float3 bn, float3 cn)
                        {

                                float3 local_trianglePos[TRIANGLE_VERTICES];
                                float3 local_triangleNor[TRIANGLE_VERTICES];


                                //=============================================================================

                                float3 _ba_ = (b - a);
                                float3 _ca_ = (c - a);


                                float3 axis[TRIANGLE_VERTICES];
                                axis[LOCAL_X] = normalize(_ba_);
                                axis[LOCAL_Z] = normalize(cross(axis[LOCAL_X], _ca_));
                                axis[LOCAL_Y] = normalize(cross(axis[LOCAL_X], axis[LOCAL_Z]));



                                float3x3 transformMatrix = (float3x3_(axis[LOCAL_X], axis[LOCAL_Y], axis[LOCAL_Z])); //set columns


                                local_trianglePos[0] = float3_(0, 0, 0);
                                local_triangleNor[0] = mul(transformMatrix, an);

                                local_trianglePos[1] = mul(transformMatrix, _ba_);
                                local_triangleNor[1] = mul(transformMatrix, bn);

                                local_trianglePos[2] = mul(transformMatrix, _ca_);
                                local_triangleNor[2] = mul(transformMatrix, cn);

                                float2 u;
                                float2 v;

                                float2 du;
```

```
                    float2 dv;

                    u.x = local_trianglePos[1].x - local_trianglePos↙
                        ↳ [0].x;
230                 u.y = local_trianglePos[2].x - local_trianglePos↙
                        ↳ [1].x;

                    v.x = local_trianglePos[1].y - local_trianglePos↙
                        ↳ [0].y;
                    v.y = local_trianglePos[2].y - local_trianglePos↙
                        ↳ [1].y;

235                 du.x = local_triangleNor[1].x - ↙
                        ↳ local_triangleNor[0].x;
                    du.y = local_triangleNor[2].x - ↙
                        ↳ local_triangleNor[1].x;

                    dv.x = local_triangleNor[1].y - ↙
                        ↳ local_triangleNor[0].y;
                    dv.y = local_triangleNor[2].y - ↙
                        ↳ local_triangleNor[1].y;
240
                    //=============================================================
                        ↳

                    //inverse of 3x3 matrix
245                 //invB = inverse of B

                    float1 aa = dot(u, u);
                    float1 bb = dot(u, v);
                    float1 cc = dot(v, v);
250
                    float1 invB_B = -(bb * cc);
                    float1 invB_C =  (bb * bb);
                    float1 invB_D =  (aa * cc);
                    float1 invB_E = -(aa * bb);
255                 float1 invB_A = -invB_C + (cc * cc) + invB_D;
                    float1 invB_F =  (aa * aa) + invB_D - invB_C;

                    float1 determinantB = 1.0 / ((aa + cc) * (invB_D↙
                        ↳  - invB_C));

260

                    //=============================================================
                        ↳

                    float4x4 tmp;
265

                    tmp[0][0] = v.x * invB_B + u.x * invB_A;
                    tmp[1][0] = v.x * invB_C + u.x * invB_B;
                    tmp[2][0] = v.y * invB_B + u.y * invB_A;
270                 tmp[3][0] = v.y * invB_C + u.y * invB_B;
```

```
                                tmp[0][1] = v.x * invB_D + u.x * invB_B;
                                tmp[1][1] = v.x * invB_E + u.x * invB_D;
                                tmp[2][1] = v.y * invB_D + u.y * invB_B;
                                tmp[3][1] = v.y * invB_E + u.y * invB_D;


                                tmp[0][2] = v.x * invB_E + u.x * invB_C;
                                tmp[1][2] = v.x * invB_F + u.x * invB_E;
                                tmp[2][2] = v.y * invB_E + u.y * invB_C;
                                tmp[3][2] = v.y * invB_F + u.y * invB_E;


                                tmp[0][3] = 0;
                                tmp[1][3] = 0;
                                tmp[2][3] = 0;
                                tmp[3][3] = 0;

                                //=============================================================================
                                    ↪

                                float4 tmp_gpu = determinantB * mul(float4_(du.x↙
                                    ↪ , dv.x, du.y, dv.y), tmp);

                                //=============================================================================
                                    ↪


                                float1 D = dot(tmp_gpu, tmp_gpu) + (3.0 * ↙
                                    ↪ tmp_gpu.y * tmp_gpu.y - 2.0 * tmp_gpu.x * ↙
                                    ↪ tmp_gpu.z);

                                if (D < 0) D = 0;
                                float1 sqrtD = sqrt(D);

                                float1 L1 = ((tmp_gpu.x + tmp_gpu.z) + sqrtD) * ↙
                                    ↪ 0.5;
                                float1 L2 = ((tmp_gpu.x + tmp_gpu.z) - sqrtD) * ↙
                                    ↪ 0.5;

                                float1 gauss = L1 * L2;

                                float1 mean = 0.5 * (L1 + L2);

                                return float2_(gauss, mean);
                            }


void curvature(const kfb *kfb, int i, int j, int xi, int xj, int yi, int yj, ↙
    ↪ float3 *normals, float2 *curves) {
  int i0 = i;
  int j0 = j;
  int ix = i + xi;
  int jx = j + xj;
```

```
  int iy = i + yi;
  int jy = j + yj;
  int w = kfb_width(kfb);
  int h = kfb_height(kfb);
  if (ix == w) {
    ix -= 2;
  }
  if (ix == -1) {
    ix += 2;
  }
  if (iy == w) {
    iy -= 2;
  }
  if (iy == -1) {
    iy += 2;
  }
  if (jx == h) {
    jx -= 2;
  }
  if (jx == -1) {
    jx += 2;
  }
  if (jy == h) {
    jy -= 2;
  }
  if (jy == -1) {
    jy += 2;
  }
  double n0 = kfb_get(kfb, i0, j0);
  double nx = kfb_get(kfb, ix, jx);
  double ny = kfb_get(kfb, iy, jy);
  if (n0 != kfb_maxiter(kfb)) {
    float3 a = float3_(i0, j0, log(n0));
    float3 b = float3_(ix, jx, log(nx));
    float3 c = float3_(iy, jy, log(ny));
    float3 an = normals[i0 * h + j0];
    float3 bn = normals[ix * h + jx];
    float3 cn = normals[iy * h + jy];
    *curves = *curves + CalcCurvature_ShapeOperator_2edge(a, b, c, an, bn, cn);
  }
}

void normal(const kfb *kfb, int i, int j, int xi, int xj, int yi, int yj, float3 *normals) {
  int i0 = i;
  int j0 = j;
  int ix = i + xi;
  int jx = j + xj;
  int iy = i + yi;
  int jy = j + yj;
  int w = kfb_width(kfb);
  int h = kfb_height(kfb);
  if (ix == w) {
    ix -= 2;
  }
  if (ix == -1) {
```

```
      ix += 2;
    }
    if (iy == w) {
      iy -= 2;
    }
    if (iy == -1) {
      iy += 2;
    }
    if (jx == h) {
      jx -= 2;
    }
    if (jx == -1) {
      jx += 2;
    }
    if (jy == h) {
      jy -= 2;
    }
    if (jy == -1) {
      jy += 2;
    }
    double n0 = kfb_get(kfb, i0, j0);
    double nx = kfb_get(kfb, ix, jx);
    double ny = kfb_get(kfb, iy, jy);
    if (n0 != kfb_maxiter(kfb)) {
      float3 a = float3_(i0, j0, log(n0));
      float3 b = float3_(ix, jx, log(nx));
      float3 c = float3_(iy, jy, log(ny));
      float3 u = normalize(b - a);
      float3 v = normalize(c - a);
      float3 w = normalize(cross(u, v));
      if (w.z < 0)
      {
        w = -w;
      }
      *normals = *normals + w;
    }
  }

int main(int argc, char **argv) {
  (void) argc;
  (void) argv;
  kfb *kfb = kfb_load(stdin);
  if (! kfb) { return 1; }
  int width = kfb_width(kfb);
  int height = kfb_height(kfb);
  pgm *pgm = pgm_alloc(width, height);
  if (! pgm) { kfb_free(kfb); return 1; }
  float3 *normals = (float3 *) calloc(1, width * height * sizeof(*normals));
  float2 *curves = (float2 *) calloc(1, width * height * sizeof(*curves));
  #pragma omp parallel for
  for (int i = 0; i < width; ++i) {
    for (int j = 0; j < height; ++j) {
      int k = i * height + j;
      normal(kfb, i, j, 1, 0, 0, 1, normals+k);
//      normal(kfb, i, j, 1, 0, 1, 1, normals+k);
//      normal(kfb, i, j, 1, 1, 0, 1, normals+k);
      normal(kfb, i, j, 1, 0, 0, -1, normals+k);
```

```
435   //      normal(kfb, i, j, 1, 0, 1, -1, normals+k);
      //      normal(kfb, i, j, 1, -1, 0, -1, normals+k);
            normal(kfb, i, j, -1, 0, 0, 1, normals+k);
      //      normal(kfb, i, j, -1, 0, -1, 1, normals+k);
      //      normal(kfb, i, j, -1, 1, 0, 1, normals+k);
440         normal(kfb, i, j, -1, 0, 0, -1, normals+k);
      //      normal(kfb, i, j, -1, 0, -1, -1, normals+k);
      //      normal(kfb, i, j, -1, -1, 0, -1, normals+k);
          }
        }
445     #pragma omp parallel for
        for (int k = 0; k < width * height; ++k) {
          normals[k] = normalize(normals[k]);
        }
        #pragma omp parallel for
450     for (int i = 0; i < width; ++i) {
          for (int j = 0; j < height; ++j) {
            int k = i * height + j;
            curvature(kfb, i, j, 1, 0, 0, 1, normals, curves+k);
            curvature(kfb, i, j, 1, 0, 0, -1, normals, curves+k);
455         curvature(kfb, i, j, -1, 0, 0, 1, normals, curves+k);
            curvature(kfb, i, j, -1, 0, 0, -1, normals, curves+k);
          }
        }
        #pragma omp parallel for
460     for (int k = 0; k < width * height; ++k) {
          curves[k] = curves[k] / 4.0;
        }
        #pragma omp parallel for
        for (int i = 0; i < width; ++i) {
465       for (int j = 0; j < height; ++j) {
            double c = curves[i * height + j].x * 100;
            if (c > 0) { c =  log(1 + c); }
            else       { c = -log(1 - c); }
            pgm_setf(pgm, i, j, 0.5 + 0.5 * c);
470       }
        }
        pgm_save(pgm, stdout);
        #pragma omp parallel for
        for (int i = 0; i < width; ++i) {
475       for (int j = 0; j < height; ++j) {
            double c = curves[i * height + j].y * 100;
            if (c > 0) { c =  log(1 + c); }
            else       { c = -log(1 - c); }
            pgm_setf(pgm, i, j, 0.5 + 0.5 * c);
480       }
        }
        pgm_save(pgm, stdout);
        free(curves);
        free(normals);
485     pgm_free(pgm);
        kfb_free(kfb);
        return 0;
      }
```

4 kfb-de-histogram.c

```c
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

#include "kfb.h"
#include "pgm.h"

int cmp_double(const void *a, const void *b) {
  const double *x = (const double *) a;
  const double *y = (const double *) b;
  double p = *x;
  double q = *y;
  if (p < q) return -1;
  if (p > q) return  1;
  return 0;
}

void histogram(const kfb *kfb, int i, int j, const double *histo, int nhisto, double density, pgm *pgm) {
  float g = 1.0f;
  double n0 = kfb_get(kfb, i, j);
  if (n0 != kfb_maxiter(kfb)) {
    double n = kfb_get_de(kfb, i, j);
    double *np = (double *) bsearch(&n, histo, nhisto, sizeof(double), cmp_double);
    g = 1.0 - pow(1.0 - (np - histo) / (double) nhisto, 1.0 / density);
  }
  pgm_setf(pgm, i, j, g);
}

int main(int argc, char **argv) {
  double density = 1.0;
  if (argc > 1) {
    density = fmax(1.0, atof(argv[1]));
  }
  kfb *kfb = kfb_load(stdin);
  if (! kfb) { return 1; }
  if (! kfb_get_hasde(kfb)) { return 1; }
  size_t width = kfb_width(kfb);
  size_t height = kfb_height(kfb);
  double *histo = (double *) malloc(width * height * sizeof(double));
  int k = 0;
  #pragma omp parallel for
  for (size_t i = 0; i < width; ++i) {
    for (size_t j = 0; j < height; ++j) {
      size_t kk;
      #pragma omp atomic capture
      kk = k++;
      histo[kk] = kfb_get_de(kfb, i, j);
    }
  }
  qsort(histo, width * height, sizeof(double), cmp_double);
  size_t nhisto;
  for (nhisto = 0; nhisto < width * height; ++nhisto) {
    if (histo[nhisto] > 0) {
      break;
    }
```

```
  }
  pgm *pgm = pgm_alloc(width, height);
  #pragma omp parallel for
  for (size_t i = 0; i < width; ++i) {
    for (size_t j = 0; j < height; ++j) {
      histogram(kfb, i, j, histo + nhisto, width * height - nhisto, density, pgm↙
          ↳ );
    }
  }
  pgm_save(pgm, stdout);
  pgm_free(pgm);
  free(histo);
  kfb_free(kfb);
  return 0;
}
```

5 kfb-expmap.c

```
#define _POSIX_C_SOURCE 200809L
// glibc feature-test macro for getline

#include <assert.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "kfb.h"

const double pi = 3.141592653589793;

struct expmap {
  FILE *in;
  char *lineptr;
  size_t linesize;
  size_t width;
  size_t height;
  int circumference;
  int radius;
  double maxiter;
  int row;
  kfb *kfb_;
};
typedef struct expmap expmap;

int expmap_load(expmap *expmap, int pass) {
  if (! expmap) {
    return 0;
  }
  if (expmap->kfb_) {
    kfb_free(expmap->kfb_);
    expmap->kfb_ = 0;
  }
  int len = getline(&expmap->lineptr, &expmap->linesize, expmap->in);
  if (len == -1) {
    expmap->lineptr = 0;
    return 0;
```

```
  }
  if (len > 0) {
    if (expmap->lineptr[len - 1] == '\n') {
      expmap->lineptr[len - 1] = 0;
    }
    FILE *kin = fopen(expmap->lineptr, "rb");
    free(expmap->lineptr);
    expmap->lineptr = 0;
    if (! kin) {
      return 0;
    }
    expmap->kfb_ = kfb_load(kin);
    fclose(kin);
    if (! expmap->kfb_) {
      return 0;
    }
    expmap->maxiter = kfb_maxiter(expmap->kfb_);
    expmap->row = 0;
    if (pass > 0) {
      if (! (expmap->width == kfb_width(expmap->kfb_) && expmap->height == kfb_height(expmap->kfb_))) {
        return 0;
      }
    }
    expmap->width = kfb_width(expmap->kfb_);
    expmap->height = kfb_height(expmap->kfb_);
    expmap->circumference = expmap->height * pi / 2;
    // 0.5 ^ (1 - 1 / radius) - 0.5 = |(0.5, 0)-(0.5*cos(t), 0.5*sin(t))| where t = 2 * pi / circumference
    double t = 2 * pi / expmap->circumference;
    double dx = 0.5 - 0.5 * cos(t);
    double dy = 0.5 * sin(t);
    double d = sqrt(dx * dx + dy * dy);
    // (1 - 1 / radius) log 0.5 = log (d + 0.5)
    // 1 - 1 / radius = log (d + 0.5) / log 0.5
    // 1 / radius = 1 - log (d + 0.5) / log 0.5
    expmap->radius = 1.0 / (1.0 - log(d + 0.5) / log(0.5));
    return 1;
  }
  return 0;
}

int expmap_get_circumference(const expmap *expmap) {
  assert(expmap);
  return expmap->circumference;
}

int expmap_get_radius(const expmap *expmap) {
  assert(expmap);
  return expmap->radius;
}

double expmap_get_maxiter(const expmap *expmap) {
  assert(expmap);
  return expmap->maxiter;
}
```

```
95  expmap *expmap_create(FILE *in) {
      if (! in) { return 0; }
      expmap *expmap = (struct expmap *) calloc(1, sizeof(struct expmap));
      if (! expmap) { return 0; }
      memset(expmap, 0, sizeof(struct expmap));
100   expmap->in = in;
      if (! expmap_load(expmap, 0)) { free(expmap); return 0; }
      return expmap;
    }

105 void expmap_free(expmap *expmap) {
      if (expmap) {
        if (expmap->lineptr) { free(expmap->lineptr); }
        if (expmap->kfb_) { kfb_free(expmap->kfb_); }
        free(expmap);
110   }
    }

    double *expmap_get_row(expmap *expmap) {
      if (! expmap) {
115     return 0;
      }
      if (expmap->row >= expmap->radius) {
        if (! expmap_load(expmap, 1)) {
          return 0;
120     }
      }
      double *row = (double *) malloc(expmap->circumference * sizeof(double));
      if (! row) {
        return 0;
125   }
      #pragma omp parallel for
      for (int k = 0; k < expmap->circumference; ++k) {
        double r = pow(0.5, 1.0 - expmap->row / (double) expmap->radius);
        double t = 2 * pi * k / (double) expmap->circumference;
130     double i = r * cos(t) * expmap->height / 2.0 + expmap->width / 2.0;
        double j = r * sin(t) * expmap->height / 2.0 + expmap->height / 2.0;
        row[k] = kfb_get_nearest(expmap->kfb_, i, j);
      }
      expmap->row++;
135   return row;
    }

    void expmap_free_row(double *row) {
      if (row) { free(row); }
140 }

    void pseudode(const double *row0, const double *row1, double maxiter0, double ↙
        ↳ maxiter1, int circumference, unsigned char *pixels) {
      (void) maxiter0;
      #pragma omp parallel for
145   for (int k = 0; k < circumference; ++k) {
        int k1 = (k + 1) % circumference;
        float g;
        double n0 = row1[k];
        if (n0 >= maxiter1) {
150       g = 1.0f;
```

```
    } else {
      double nx = row1[k1];
      double ny = row0[k];
      double zx = nx - n0;
      double zy = ny - n0;
      double zz = 1.0;
      double z = sqrt(zx * zx + zy * zy + zz * zz);
      g = zz / z;
    }
    pixels[k] = fminf(fmaxf(255.0f * g, 0.0f), 255.0f);
  }
}

int main(int argc, char **argv) {
  int count;
  if (argc > 1) {
    count = atoi(argv[1]);
    if (! (count > 0)) { return 1; }
  } else {
    return 1;
  }
  expmap *expmap = expmap_create(stdin);
  if (! expmap) { return 1; }
  int circumference = expmap_get_circumference(expmap);
  int radius = expmap_get_radius(expmap);
  int width = circumference;
  int height = radius * count - 1; // -1 for pseudo-de colouring
  fprintf(stderr, "width: %d\nheight: %d per kfb\nheight: %d total\n", ↙
      ↳ circumference, radius, height);
  unsigned char *pixels = (unsigned char *) malloc(width);
  if (! pixels) {
    expmap_free(expmap);
    return 1;
  }
  fprintf(stdout, "P5\n%d %d\n255\n", width, height);
  double *rows[2];
  rows[0] = 0;
  rows[1] = expmap_get_row(expmap);
  double maxiter[2];
  maxiter[0] = 0;
  maxiter[1] = expmap_get_maxiter(expmap);
  while (1) {
    if (rows[0]) { expmap_free_row(rows[0]); }
    rows[0] = rows[1];
    rows[1] = expmap_get_row(expmap);
    if (! rows[1]) { break; }
    maxiter[0] = maxiter[1];
    maxiter[1] = expmap_get_maxiter(expmap);
    pseudode(rows[0], rows[1], maxiter[0], maxiter[1], circumference, pixels);
    fwrite(pixels, width, 1, stdout); // FIXME check success
  }
  if (rows[0]) { expmap_free_row(rows[0]); }
  expmap_free(expmap);
  free(pixels);
  return 0;
}
```

6 kfb.h

```
#ifndef KFB_H
#define KFB_H 1

#include <stdio.h>

struct kfb;
typedef struct kfb kfb;

extern kfb *kfb_load(FILE *in);
extern kfb *kfb_alloc(const kfb *proto, size_t width, size_t height);
extern void kfb_free(kfb *k);
extern int kfb_save(const kfb *kfb, FILE *out);
extern size_t kfb_width(const kfb *kfb);
extern size_t kfb_height(const kfb *kfb);
extern double kfb_maxiter(const kfb *kfb);
extern double kfb_get(const kfb *kfb, size_t i, size_t j);
extern double kfb_get_nearest(const kfb *kfb, double i, double j);
extern double kfb_get_linear(const kfb *kfb, double i, double j);
extern double kfb_get_cubic(const kfb *kfb, double i, double j);
extern void kfb_set(kfb *kfb, size_t i, size_t j, double n);

extern int kfb_get_hasde(const kfb *kfb);
extern void kfb_set_hasde(kfb *kfb, int hasde);
extern double kfb_get_de(const kfb *kfb, size_t i, size_t j);
extern void kfb_set_de(kfb *kfb, size_t i, size_t j, double de);

extern int   kfb_get_raw_count(const kfb *kfb, size_t i, size_t j);
extern float kfb_get_raw_trans(const kfb *kfb, size_t i, size_t j);
extern float kfb_get_raw_de   (const kfb *kfb, size_t i, size_t j);

#endif
```

7 kfb-histogram.c

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

#include "kfb.h"
#include "pgm.h"

int cmp_double(const void *a, const void *b) {
  const double *x = (const double *) a;
  const double *y = (const double *) b;
  double p = *x;
  double q = *y;
  if (p < q) return -1;
  if (p > q) return  1;
  return 0;
}

void histogram(const kfb *kfb, int i, int j, const double *histo, int nhisto, ↙
    ↳ double density, pgm *pgm) {
  float g = 0.0f;
  double n0 = kfb_get(kfb, i, j);
```

```
  if (n0 != kfb_maxiter(kfb)) {
    double *np = (double *) bsearch(&n0, histo, nhisto, sizeof(double), ↙
        ↳ cmp_double);
    g = 1.0 - pow(1.0 - (np - histo) / (double) nhisto, 1.0 / density);
  }
  pgm_setf(pgm, i, j, g);
}

int main(int argc, char **argv) {
  double density = 1.0;
  if (argc > 1) {
    density = fmax(1.0, atof(argv[1]));
  }
  kfb *kfb = kfb_load(stdin);
  if (! kfb) { return 1; }
  int width = kfb_width(kfb);
  int height = kfb_height(kfb);
  int maxiter = kfb_maxiter(kfb);
  double *histo = (double *) malloc(width * height * sizeof(double));
  int k = 0;
  #pragma omp parallel for
  for (int i = 0; i < width; ++i) {
    for (int j = 0; j < height; ++j) {
      int kk;
      #pragma omp atomic capture
      kk = k++;
      histo[kk] = kfb_get(kfb, i, j);
    }
  }
  qsort(histo, width * height, sizeof(double), cmp_double);
  int nhisto;
  for (nhisto = width * height; nhisto > 0; --nhisto) {
    if (histo[nhisto-1] < maxiter) {
      break;
    }
  }
  pgm *pgm = pgm_alloc(width, height);
  #pragma omp parallel for
  for (int i = 0; i < width; ++i) {
    for (int j = 0; j < height; ++j) {
      histogram(kfb, i, j, histo, nhisto, density, pgm);
    }
  }
  pgm_save(pgm, stdout);
  pgm_free(pgm);
  free(histo);
  kfb_free(kfb);
  return 0;
}
```

8 kfb-pseudo-de.c

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

#include "kfb.h"
```

```c
#include "pgm.h"

void pseudode(const kfb *kfb, size_t i, size_t j, pgm *pgm) {
  size_t i0 = i;
  size_t j0 = j;
  size_t ix = i + 1;
  size_t jx = j;
  int sx = 1;
  size_t iy = i;
  size_t jy = j + 1;
  int sy = 1;
  if (ix == kfb_width(kfb)) {
    ix -= 2;
    sx = -1;
  }
  if (jy == kfb_height(kfb)) {
    jy -= 2;
    sx = -1;
  }
  double n0 = kfb_get(kfb, i0, j0);
  float g;
  if (n0 == kfb_maxiter(kfb)) {
    g = 1.0f;
  } else {
    double nx = kfb_get(kfb, ix, jx);
    double ny = kfb_get(kfb, iy, jy);
    double zx = sx * (nx - n0);
    double zy = sy * (ny - n0);
    double zz = 1.0;
    double z = sqrt(zx * zx + zy * zy + zz * zz);
    g = zz / z;
  }
  pgm_setf(pgm, i, j, g);
}

int main(int argc, char **argv) {
  (void) argc;
  (void) argv;
  kfb *kfb = kfb_load(stdin);
  if (! kfb) { return 1; }
  size_t width = kfb_width(kfb);
  size_t height = kfb_height(kfb);
  pgm *pgm = pgm_alloc(width, height);
  if (! pgm) { kfb_free(kfb); return 1; }
  #pragma omp parallel for
  for (size_t i = 0; i < width; ++i) {
    for (size_t j = 0; j < height; ++j) {
      pseudode(kfb, i, j, pgm);
    }
  }
  pgm_save(pgm, stdout);
  pgm_free(pgm);
  kfb_free(kfb);
  return 0;
}
```

9 kfb-rainbow.c

```
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

#include "kfb.h"

const double pi = 3.141592653589793;

void hsv2rgb(float h, float s, float v, float *red, float *grn, float *blu) {
  float i, f, p, q, t, r, g, b;
  if (s == 0) { r = g = b = v; } else {
    h = 6 * (h - floorf(h));
    int ii = i = floorf(h);
    f = h - i;
    p = v * (1 - s);
    q = v * (1 - (s * f));
    t = v * (1 - (s * (1 - f)));
    switch(ii) {
      case 0: r = v; g = t; b = p; break;
      case 1: r = q; g = v; b = p; break;
      case 2: r = p; g = v; b = t; break;
      case 3: r = p; g = q; b = v; break;
      case 4: r = t; g = p; b = v; break;
      default:r = v; g = p; b = q; break;
    }
  }
  *red = r;
  *grn = g;
  *blu = b;
}

void rainbow(const kfb *kfb, size_t i, size_t j, unsigned char *pixel) {
  size_t i0 = i;
  size_t j0 = j;
  size_t ix = i + 1;
  size_t jx = j;
  int sx = 1;
  size_t iy = i;
  size_t jy = j + 1;
  int sy = 1;
  if (ix == kfb_width(kfb)) {
    ix -= 2;
    sx = -1;
  }
  if (jy == kfb_height(kfb)) {
    jy -= 2;
    sx = -1;
  }
  double n0 = kfb_get(kfb, i0, j0);
  float r, g, b;
  if (n0 == kfb_maxiter(kfb)) {
    r = g = b = 1.0f;
  } else {
    double nx = kfb_get(kfb, ix, jx);
    double ny = kfb_get(kfb, iy, jy);
```

```
      double zx = sx * (nx - n0);
      double zy = sy * (ny - n0);
      double zz = 1.0;
      double z = sqrt(zx * zx + zy * zy + zz * zz);
      zx /= z;
      zy /= z;
      zz /= z;
      double hue = atan2(zy, zx) / (2 * pi);
      double sat = 0.5 * zz;
      double val = zz;
      hsv2rgb(hue, sat, val, &r, &g, &b);
    }
    size_t k = (kfb_width(kfb) * j + i) * 3;
    pixel[k+0] = fminf(fmaxf(255 * r, 0), 255);
    pixel[k+1] = fminf(fmaxf(255 * g, 0), 255);
    pixel[k+2] = fminf(fmaxf(255 * b, 0), 255);
  }

  int main(int argc, char **argv) {
    (void) argc;
    (void) argv;
    kfb *kfb = kfb_load(stdin);
    if (! kfb) { return 1; }
    size_t width = kfb_width(kfb);
    size_t height = kfb_height(kfb);
    unsigned char *pixels = (unsigned char *) malloc(width * height * 3);
    #pragma omp parallel for
    for (size_t i = 0; i < width; ++i) {
      for (size_t j = 0; j < height; ++j) {
        rainbow(kfb, i, j, pixels);
      }
    }
    fprintf(stdout, "P6\n%ld %ld\n255\n", width, height);
    fwrite(pixels, width * height * 3, 1, stdout);
    free(pixels);
    kfb_free(kfb);
    return 0;
  }
```

10 kfb-resize.c

```
  #include <math.h>
  #include <stdio.h>
  #include <stdlib.h>

  #include "kfb.h"
  #include "pgm.h"

  int main(int argc, char **argv) {
    int owidth;
    int oheight;
    if (argc > 2) {
      owidth = atoi(argv[1]);
      oheight = atoi(argv[2]);
    } else {
      return 1;
    }
```

```
  if (! (owidth > 0 && oheight > 0)) {
    return 1;
  }
  kfb *in = kfb_load(stdin);
  if (! in) { return 1; }
  int iwidth = kfb_width(in);
  int iheight = kfb_height(in);
  kfb *out = kfb_alloc(in, owidth, oheight);
  if (! out) { kfb_free(in); return 1; }
  double sx = iwidth / (double) owidth;
  double sy = iheight / (double) oheight;
  #pragma omp parallel for
  for (int i = 0; i < owidth; ++i) {
    for (int j = 0; j < oheight; ++j) {
      kfb_set(out, i, j, kfb_get_cubic(in, sx * i, sy * j));
    }
  }
  kfb_save(out, stdout);
  kfb_free(out);
  kfb_free(in);
  return 0;
}
```

11 kfb-statistics.c

```
#include <assert.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

#include "kfb.h"

int cmp_double(const void *a, const void *b) {
  const double *x = (const double *) a;
  const double *y = (const double *) b;
  double p = *x;
  double q = *y;
  if (p < q) return -1;
  if (p > q) return  1;
  return 0;
}

int main(int argc, char **argv) {
  (void) argc;
  (void) argv;
  kfb *kfb = kfb_load(stdin);
  if (! kfb) { return 1; }
  int width = kfb_width(kfb);
  int height = kfb_height(kfb);
  int maxiter = kfb_maxiter(kfb);
  double *histo = (double *) malloc(width * height * sizeof(double));
  int k = 0;
  #pragma omp parallel for
  for (int i = 0; i < width; ++i) {
    for (int j = 0; j < height; ++j) {
      int kk;
      #pragma omp atomic capture
```

```
        kk = k++;
        histo[kk] = kfb_get(kfb, i, j);
      }
    }
    qsort(histo, width * height, sizeof(double), cmp_double);
    int nhisto;
    for (nhisto = width * height; nhisto > 0; --nhisto) {
      if (histo[nhisto-1] < maxiter) {
        break;
      }
    }
    int mhisto;
    for (mhisto = 0; mhisto < nhisto; ++mhisto)
    {
      if (histo[mhisto] >= 0) {
        break;
      }
    }
    printf("%3.6f%%\t%d glitched pixels\n", mhisto * 100.0 / (width * height), ↙
      ↪ mhisto);
    printf("%3.6f%%\t%d unescaped pixels\n", (width * height - nhisto) * 100.0 / (↙
      ↪ width * height), width * height - nhisto);
    printf("percentile\titerations\n");
    for (k = 0; k <= 10; ++k) {
      int l = round(mhisto + k / 10.0 * (nhisto - 1 - mhisto));
      assert(0 <= l);
      assert(l < width * height);
      printf("%12d%%\t%.3f\n", 10 * k, histo[l]);
    }
    free(histo);
    kfb_free(kfb);
    return 0;
  }
```

12 kfb-stretch.c

```
  #include <math.h>
  #include <stdio.h>
  #include <stdlib.h>

  #include "kfb.h"
  #include "pgm.h"

  void stretch(const kfb *kfb, int i, int j, double nmin, double nmax, pgm *pgm) {
    float g = 0.0f;
    double n = kfb_get(kfb, i, j);
    if (n != kfb_maxiter(kfb)) {
      g = (n - nmin) / (nmax - nmin);
    }
    pgm_setf(pgm, i, j, g);
  }

  int main(int argc, char **argv) {
    (void) argc;
    (void) argv;
    kfb *kfb = kfb_load(stdin);
    if (! kfb) { return 1; }
```

```
  int width = kfb_width(kfb);
  int height = kfb_height(kfb);
  double maxiter = kfb_maxiter(kfb);
  double nmax = -1.0 / 0.0;
  double nmin =  1.0 / 0.0;
  for (int i = 0; i < width; ++i) {
    for (int j = 0; j < height; ++j) {
      double n = kfb_get(kfb, i, j);
      if (n != maxiter) {
        nmax = fmax(n, nmax);
        nmin = fmin(n, nmin);
      }
    }
  }
  pgm *pgm = pgm_alloc(width, height);
  if (! pgm) { kfb_free(kfb); return 1; }
  #pragma omp parallel for
  for (int i = 0; i < width; ++i) {
    for (int j = 0; j < height; ++j) {
      stretch(kfb, i, j, nmin, nmax, pgm);
    }
  }
  pgm_save(pgm, stdout);
  pgm_free(pgm);
  kfb_free(kfb);
  return 0;
}
```

13 kfb-to-exr.cc

```
#include <cstdio>
#include <cstdlib>
#include <iostream>

#include <ImfNamespace.h>
#include <ImfOutputFile.h>
#include <ImfIntAttribute.h>
#include <ImfChannelList.h>

#include "kfb.h"

namespace IMF = OPENEXR_IMF_NAMESPACE;
using namespace IMF;

bool writeEXR(const char *filename, int width, int height, int maxiter, const ↙
    ↳ uint32_t *n, const float *nf, const float *de)
{
  if (n == nullptr && nf == nullptr && de == nullptr) return false;
  Header header(width, height);
  header.insert("Iterations", IntAttribute(maxiter));
  if (n  != nullptr) header.channels().insert("N",  Channel(IMF::UINT));
  if (nf != nullptr) header.channels().insert("NF", Channel(IMF::FLOAT));
  if (de != nullptr) header.channels().insert("DE", Channel(IMF::FLOAT));
  OutputFile of(filename, header);
  FrameBuffer fb;
  if (n  != nullptr) fb.insert("N",  Slice(IMF::UINT,  (char *) n,  sizeof(*n ) ↙
      ↳ * 1, sizeof(*n ) * width));
```

```
  if (nf != nullptr) fb.insert("NF", Slice(IMF::FLOAT, (char *) nf, sizeof(*nf) ↙
      ↳ * 1, sizeof(*nf) * width));
  if (de != nullptr) fb.insert("DE", Slice(IMF::FLOAT, (char *) de, sizeof(*de) ↙
      ↳ * 1, sizeof(*de) * width));
  of.setFrameBuffer(fb);
  of.writePixels(height);
  return true;
}

int main(int argc, char **argv) {
  if (argc < 3) return 1;
  std::FILE *in = std::fopen(argv[1], "rb");
  if (! in) return 1;
  kfb *kfb = kfb_load(in);
  if (! kfb) { return 1; }
  int width = kfb_width(kfb);
  int height = kfb_height(kfb);
  int maxiter = kfb_maxiter(kfb);
  uint32_t *n  = (uint32_t *) std::malloc(sizeof(*n ) * width * height);
  float    *nf = (float    *) std::malloc(sizeof(*nf) * width * height);
  float    *de = kfb_get_hasde(kfb)
                 ? (float    *) std::malloc(sizeof(*de) * width * height)
                 : nullptr;
  size_t glitched = 0;
  #pragma omp parallel for reduction(+:glitched)
  for (int j = 0; j < height; ++j) {
    for (int i = 0; i < width; ++i) {
      size_t k = (size_t) j * width + i;
      int   count = kfb_get_raw_count(kfb, i, j);
      float trans = kfb_get_raw_trans(kfb, i, j);
      float dist  = de ? kfb_get_raw_de(kfb, i, j) : 0.0f;
      if (count == maxiter)
      {
        // unescaped
        n[k] = 0xFFffFFffU;
        nf[k] = 0.0f;
        if (de) de[k] = 0.0f;
      }
      else if (trans > 1.0f)
      {
        // glitch
        glitched += 1;
        n[k] = count;
        nf[k] = 0.0f;
        if (de) de[k] = 0.0f;
      }
      else
      {
        // escaped, ok
        n[k] = count;
        nf[k] = 1.0f - trans;
        if (de) de[k] = dist;
      }
    }
  }
  if (glitched)
  {
```

```
    std::cerr << argv[0] << ": WARNING " << glitched << "/" << ((size_t) width *↙
        ↪ height) << " pixels are glitched!" << std::endl;
  }
  bool ok = writeEXR(argv[2], width, height, maxiter, n, nf, de);
  return ! ok;
}
```

14 kfb-to-mmit.c

```
#include <stdio.h>
#include <stdlib.h>

#include "kfb.h"

int main(int argc, char **argv) {
  (void) argc;
  (void) argv;
  kfb *kfb = kfb_load(stdin);
  if (! kfb) { return 1; }
  int width = kfb_width(kfb);
  int height = kfb_height(kfb);
  double *mmit = (double *) malloc(width * height * sizeof(double));
  if (! mmit) { kfb_free(kfb); return 1; }
  #pragma omp parallel for
  for (int j = 0; j < height; ++j) {
    for (int i = 0; i < width; ++i) {
      int k = j * width + i;
      mmit[k] = kfb_get(kfb, i, j);
    }
  }
  // FIXME check sizeof(int) == 4, little-endian
  // FIXME check sizeof(double) == 8, little-endian, IEEE
  // FIXME check write success
  fwrite(&width, sizeof(width), 1, stdout);
  fwrite(&height, sizeof(height), 1, stdout);
  fwrite(mmit, width * height * sizeof(double), 1, stdout);
  free(mmit);
  kfb_free(kfb);
  return 0;
}
```

15 kfb-to-ogv.sh

```
#!/bin/bash

# change these lines to suit your environment
zoom="${HOME}/code/maximus_book/code/zoom"

# change these lines to suit your taste
sr=8000
aa=2

seconds="${1}"

work="${2}"
if [ "${work}x" == "x" ]
then
```

```
    work="$(pwd)"
  fi

  tmp="${3}"
  if [ "${tmp}x" == "x" ]
  then
    tmp="${work}"
  fi

  echo "checking required programs"
  (
  which kfb-pseudo-de || echo "FAIL kfb-pseudo-de"
  which kfb-expmap || echo "FAIL kfb-expmap"
  test -x "${zoom}" || echo "FAIL zoom (book)"
  which pnmscale || echo "FAIL pnmscale"
  which pgmtoppm || echo "FAIL pgmtoppm"
  which pnmflip || echo "FAIL pnmflip"
  which ecasound || echo "FAIL ecasound"
  which oggenc || echo "FAIL oggenc"
  which avconv || echo "FAIL avconv"
  ) | tee "${tmp}/programs.log"
  if grep -q "^FAIL" "${tmp}/programs.log"
  then
    echo "some programs missing, aborting"
    exit 1
  fi

  pushd "${work}"

  count="$(ls *.kfb | wc -l)"
  echo "${count} kfb files"

  echo "converting kfb to ppm"
  for i in *.kfb
  do
    kfb-pseudo-de < "${i}" | pnmscale -reduce "${aa}" 2>/dev/null | pgmtoppm white↙
        ↳ > "${tmp}/${i%kfb}ppm"
  done

  echo "generating exponential map"
  ls *.kfb | kfb-expmap "${count}" > "${tmp}/expmap.pgm"

  head -n 2 "${tmp}/expmap.pgm" | tail -n 1 | ( read ewidth eheight
  echo "exponential map is ${ewidth}x${eheight}"

  echo "scaling and flipping map"
  pnmscale -pixels "$(( seconds * sr ))" < "${tmp}/expmap.pgm" | pnmflip -tb > "${↙
      ↳ tmp}/expmap2.pgm"

  echo "converting map image to audio"
  cat "${tmp}/expmap2.pgm" | ecasound -f:u8,1,"${sr}" -i:stdin -f:f32,1,"${sr}" -o↙
      ↳ :"${tmp}/audio1.wav"
  echo "resampling to output sample rate"
  ecasound -f:f32,1,48000 -i:resample-hq,auto,"${tmp}/audio1.wav" -o:"${tmp}/↙
      ↳ audio2.wav"
  echo "correcting dc offset"
  ecasound -f:f32,1,48000 -i:"${tmp}/audio2.wav" -efh:1 -eaw:33,100 -o:"${tmp}/↙
```

```
    ↪ audio3.wav"
echo "applying reverb (left)"
ecasound -f:f32,1,48000 -i:"${tmp}/audio3.wav" -el:gverb↙
    ↪ ,10.1,5,0.5,0.5,-70,0,-12 -eaw:33,100 -o:"${tmp}/audio3l.wav"
echo "applying reverb (right)"
ecasound -f:f32,1,48000 -i:"${tmp}/audio3.wav" -el:gverb,9.9,5,0.5,0.5,-70,0,-12↙
    ↪  -eaw:33,100 -o:"${tmp}/audio3r.wav"
echo "combining audio tracks to stereo"
ecasound -a:1 -f:f32,1,48000 -i:"${tmp}/audio3.wav" -chcopy:1,2 \
         -a:2 -f:f32,1,48000 -i:"${tmp}/audio3l.wav" \
         -a:3 -f:f32,1,48000 -i:"${tmp}/audio3r.wav" -chmove:1,2 \
         -a:all -f:f32,2,48000 -o:"${tmp}/audio.wav"
echo "encoding audio"
oggenc -b 192 "${tmp}/audio.wav"
aseconds="$(( $(stat --format=%s "${tmp}/audio.wav") / (4 * 2 * 48000) ))"
echo "audio length is ${aseconds}"

cat "${tmp}/00001"*.ppm | head -n 2 | tail -n 1 | ( read fwidth fheight
echo "video source frames are ${fwidth}x${fheight}"

echo "encoding video"
ls "${tmp}/"*.ppm | tac | xargs cat | "${zoom}" "${fwidth}" "${fheight}" "${↙
    ↪ count}" "${aseconds}" |
  avconv -f yuv4mpegpipe -i - -i "${tmp}/audio.ogg" -acodec copy -vb 5M -↙
      ↪ shortest "${tmp}/video.ogv"

))

echo "done!"
ls -lsh "${tmp}/video.ogv"

popd
```

16 Makefile

```
EXES = \
        kfb-curvature \
        kfb-de-histogram \
        kfb-expmap \
        kfb-histogram \
        kfb-pseudo-de \
        kfb-rainbow \
        kfb-resize \
        kfb-statistics \
        kfb-stretch \
        kfb-to-mmit \
        kfb-to-exr

SCRIPTS = \
        kfb-to-ogv.sh

all: $(EXES)

clean:
        -rm $(EXES)

prefix ?= $(HOME)
```

```
install: $(EXES) $(SCRIPTS)
	cp -avf $(EXES) $(SCRIPTS) $(prefix)/bin/

.SUFFIXES:
.PHONY: all clean install

%: %.o kfb.o pgm.o
	g++ -std=c++17 -Wall -Wextra -pedantic -march=native -O3 -fopenmp -o $@ ↙
	    ↳ $< kfb.o pgm.o -lm -lggdb

%.o: %.c kfb.h pgm.h
	g++ -std=c++17 -Wall -Wextra -pedantic -march=native -O3 -fopenmp -o $@ ↙
	    ↳ -c $< -lm -lggdb

%.o: %.cc kfb.h pgm.h
	g++ -std=c++17 -Wall -Wextra -pedantic -march=native -O3 -fopenmp -o $@ ↙
	    ↳ -c $< -lm -lggdb

kfb-to-exr: kfb-to-exr.cc kfb.h kfb.o
	g++ -std=c++11 -Wall -Wextra -pedantic -Wno-deprecated -Wno-unused-↙
	    ↳ parameter -march=native -O3 -fopenmp -o $@ $< kfb.o `pkg-config --↙
	    ↳ cflags --libs OpenEXR` -lm -lggdb
```

17 pgm.c

```
#include <assert.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>

#include "pgm.h"

struct pgm {
  int width;
  int height;
  unsigned char *pixels;
};

extern pgm *pgm_alloc(int width, int height) {
  if (! (width > 0 && height > 0)) { return 0; }
  pgm *p = (pgm *) calloc(1, sizeof(struct pgm));
  if (! p) { return 0; }
  p->width = width;
  p->height = height;
  p->pixels = (unsigned char *) malloc(width * height);
  if (! p->pixels) {
    free(p);
    return 0;
  }
  return p;
}

extern void pgm_free(pgm *pgm) {
  if (pgm) {
    if (pgm->pixels) { free(pgm->pixels); }
    free(pgm);
```

```
  }
}

extern int pgm_width(const pgm *pgm) {
  assert(pgm);
  return pgm->width;
}

extern int pgm_height(const pgm *pgm) {
  assert(pgm);
  return pgm->height;
}

extern void pgm_set(pgm *pgm, int i, int j, int g) {
  assert(pgm);
  assert(pgm->pixels);
  assert(0 <= i && i < pgm->width);
  assert(0 <= j && j < pgm->height);
  int k = j * pgm->width + i;
  pgm->pixels[k] = g;
}

extern void pgm_setf(pgm *pgm, int i, int j, float g) {
  pgm_set(pgm, i, j, fminf(fmaxf(255 * g, 0), 255));
}

extern int pgm_save(const pgm *pgm, FILE *out) {
  assert(pgm);
  assert(pgm->pixels);
  fprintf(out, "P5\n%d %d\n255\n", pgm->width, pgm->height);
  return 1 == fwrite(pgm->pixels, pgm->width * pgm->height, 1, out);
}
```

18 pgm.h

```
#ifndef PGM_H
#define PGM_H 1

#include <stdio.h>

struct pgm;
typedef struct pgm pgm;

extern pgm *pgm_alloc(int width, int height);
extern void pgm_free(pgm *pgm);
extern int pgm_width(const pgm *pgm);
extern int pgm_height(const pgm *pgm);
extern void pgm_set(pgm *pgm, int i, int j, int g);
extern void pgm_setf(pgm *pgm, int i, int j, float g);
extern int pgm_save(const pgm *pgm, FILE *out);

#endif
```

19 README

```
kf-extras -- extras for manipulating output from Kalles Fraktaler 2
(C) 2014,2018 Claude Heiland-Allen, partially based on work by Karl Runmo
```

```
License: attribution, with any modified source made available (no warranty)

KF2 can be obtained from http://www.chillheimer.de/kallesfraktaler/
Discussion forum: http://www.fractalforums.com/kalles-fraktaler/

Programs included (run 'make' to build them all using 'gcc'):

expmap n < x.txt > x.pgm    # exponential map (log polar) with pseudo-de colour
                              x.txt lists kfb files one per line, in order
                              n is number of kfb files listed in x.txt
histogram d < x.kfb > x.pgm # histogram equalize (sort and search, O(N log N))
                              d is density >= 1.0, affects the brightness curve
kfbtommit < x.kfb > x.mmit  # convert to Mandel Machine iteration data format
pseudo-de < x.kfb > x.pgm   # pseudo distance estimate colouring
rainbow < x.kfb > x.ppm     # rainbow based on slope direction, plus pseudo-de
resize w h < x.kfb > x.kfb  # resize to new (w,h) using bi-cubic interpolation
stretch < x.kfb > x.pgm     # simple iteration count stretching

Scripts included ('bash' shell and common linux utilities required):

kfbtoogv.sh s work tmp      # automatic movie maker with sound generation
                              s is desired movie length in seconds
                              work is absolute path to dir containing *.kfb
                              tmp is absolute path to dir for output
                              it checks for the less-common programs it needs

--
claude@mathr.co.uk
https://mathr.co.uk
```