fractal-bits

Claude Heiland-Allen

2012–2019

Contents

1 buddhabrot/bb.c

```
// gcc -std=c99 -Wall -pedantic -Wextra -O3 -fopenmp -lm -o bb bb.c
// ./bb | pnmsplit - %d.pgm

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

typedef unsigned char B;
typedef unsigned int N;
typedef int Z;
typedef double R;
typedef double _Complex C;

#define S 1024
R accum[S][S];
B img[S][S];

static inline C to_screen(C x)
{
  return S * (0.2 * x + (0.5 + I * 0.5));
}
```

```
static inline C from_screen(C x)
{
  return (x / S - (0.5 + I * 0.5)) * 5.0;
}

static inline R cabs2(C z) { return creal(z) * creal(z) + cimag(z) * cimag(z); }

static void clear()
{
  #pragma omp parallel for
  for (Z y = 0; y < S; ++y)
    for (Z x = 0; x < S; ++x)
      accum[y][x] = 0;
}

static inline void plot(Z x, Z y, R f)
{
  if (! (isnan(f) || isinf(f)))
  {
    #pragma omp atomic update
    accum[y][x] += f;
  }
}

static void post()
{
  R m = 0;
  for (Z y = 0; y < S/2; ++y)
    for (Z x = 0; x < S; ++x)
      m = fmax(m, accum[y][x] + accum[S-1-y][x]);
  fprintf(stderr, "%.18f\n", m);
  m = 255 / m;
  #pragma omp parallel for
  for (Z y = 0; y < S/2; ++y)
    for (Z x = 0; x < S; ++x)
      img[y][x] = m * (accum[y][x] + accum[S-1-y][x]);
  #pragma omp parallel for
  for (Z y = 0; y < S/2; ++y)
    for (Z x = 0; x < S; ++x)
      img[S-1-y][x] = img[y][x];
}

static void save()
{
  fprintf(stdout, "P5\n%d %d\n255\n", S, S);
  fwrite(&img[0][0], S * S, 1, stdout);
  fflush(stdout);
}

static inline R cross(C a, C b)
```

```
{
  return creal(a) * cimag(b) - cimag(a) * creal(b);
}

static inline N inside(C a, C b, C c)
{
  return cross(a - b, c - b) < 0;
}

// https://en.wikipedia.org/wiki/Line%E2%80%93line_intersection#Given_two_points_on_each_line
static inline C intersect(C a, C b, C c, C d)
{
  R x1 = creal(a);
  R y1 = cimag(a);
  R x2 = creal(b);
  R y2 = cimag(b);
  R x3 = creal(c);
  R y3 = cimag(c);
  R x4 = creal(d);
  R y4 = cimag(d);
  R x = (x1 * y2 - y1 * x2) * (x3 - x4) - (x1 - x2) * (x3 * y4 - y3 * x4);
  R y = (x1 * y2 - y1 * x2) * (y3 - y4) - (y1 - y2) * (x3 * y4 - y3 * x4);
  R z = (x1 - x2) * (y3 - y4) - (y1 - y2) * (x3 - x4);
  if (z == 0)
    return a;
  return (x + I * y) / z;
}

// https://en.wikipedia.org/wiki/Sutherland%E2%80%93Hodgman_algorithm#Pseudo_code
static R coverage(C a, C b, C c)
{
  C clip_polygon[3][2] = { { a, b }, { b, c }, { c, a } };
  R nan = 0.0 / 0.0;
  C subject[2][16] =
    { { 0, 1, 1 + I, I,     nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan }
    , { nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan, nan }
    };
  N count = 4;
  N source_count = 4;
  N source = 1;
  for (N edge = 0; edge < 3; ++edge)
  {
    source = 1 - source;
    source_count = count;
    count = 0;
    C e0 = clip_polygon[edge][0];
    C e1 = clip_polygon[edge][1];
    C s = 0;
    if (source_count > 0)
      s = subject[source][source_count - 1];
```

```
    for (N i = 0; i < source_count; ++i)
    {
      C e = subject[source][i];
      if (inside(e, e0, e1))
      {
        if (! inside(s, e0, e1))
          subject[1-source][count++] = intersect(s, e, e0, e1);
        subject[1-source][count++] = e;
      }
      else if (inside(s, e0, e1))
        subject[1-source][count++] = intersect(s, e, e0, e1);
      s = e;
    }
  }
  source = 1 - source;
  // clipped polgon in subject[source][[0..count)]
  R cover = 0;
  C p = 0.5 + 0.5 * I;
  C s = 0;
  if (count > 0)
    s = subject[source][count - 1];
  for (N i = 0; i < count; ++i)
  {
    C e = subject[source][i];
    cover += cross(e - p, s - p);
    s = e;
  }
  return fabs(0.5 * cover);
}


// http://www.sunshine2k.de/coding/java/TriangleRasterization/↙
    ↳ TriangleRasterization.html
static void rasterize(C a, C b, C c, R a0)
{
  R r = 16;
  if (cabs2(a) < r || cabs2(b) < r || cabs2(c) < r)
  {
    R a1 = cross(b - a, c - a)*0.5;
    R f = a0 / a1;
    a = to_screen(a);
    b = to_screen(b);
    c = to_screen(c);
    C AB = b - a;
    C AC = c - a;
    R ABC = 1 / cross(AB, AC);
    Z x0 = floor(fmin(fmin(creal(a), creal(b)), creal(c)));
    Z x1 = ceil (fmax(fmax(creal(a), creal(b)), creal(c)));
    Z y0 = floor(fmin(fmin(cimag(a), cimag(b)), cimag(c)));
    Z y1 = ceil (fmax(fmax(cimag(a), cimag(b)), cimag(c)));
    if (x0 == x1 && y0 == y1)
    {
      // small triangle entirely inside pixel
      if (0 <= x0 && x0 < S && 0 <= y0 && y0 < S)
        plot(x0, y0, f * 2.0 / ABC);
    }
    else
```

```
    {
      x0 = fmax(x0, 0);
      x1 = fmin(x1, S - 1);
      y0 = fmax(y0, 0);
      y1 = fmin(y1, S - 1);
      for (Z y = y0; y <= y1; ++y)
        for (Z x = x0; x <= x1; ++x)
        {
          Z inside = 1;
          for (Z dy = 0; dy <= 1; ++dy)
            for (Z dx = 0; dx <= 1; ++dx)
            {
              C P = (x + dx) + I * (y + dy) - a;
              R s = cross(P, AC) * ABC;
              R t = cross(AB, P) * ABC;
              Z v = (s >= 0 && t >= 0 && s + t <= 1);
              inside &= v;
            }
          if (inside)
            // pixel entirely inside triangle
            plot(x, y, f);
          else
          {
            C p = x + I * y;
            R g = coverage(a - p, b - p, c - p);
            plot(x, y, f * g);
          }
        }
    }
  }
}

static void plot_iterates(C c0, R dc, N n) {
  C z[4] = { 0, 0, 0, 0 };
  C c[4] = { c0 - dc - I * dc, c0 + dc - I * dc, c0 + dc + I * dc, c0 - dc + I * dc };
  R a0 = dc * dc * 0.5;
  for (N i = 0; i < n; ++i)
  {
    for (N k = 0; k < 4; ++k)
      z[k] = z[k] * z[k] + c[k];
    rasterize(z[0], z[1], z[2], a0);
    rasterize(z[0], z[2], z[3], a0);
  }
}


static int attractor(C *z0, C c, N period) {
  R eps = 1e-12;
  R er2 = 16;
  C z = *z0;
  for (N j = 0; j < 256; ++j) {
    C dz = 1.0;
    for (N k = 0; k < period; ++k) {
      dz = 2.0 * dz * z;
      z = z * z + c;
```

```
    }
    R z2 = cabs(z);
    if (! (z2 < er2)) {
      break;
    }
    z = *z0 - (z - *z0) / (dz - 1.0);
    R e = cabs(z - *z0);
    *z0 = z;
    if (e < eps) {
      return 1;
    }
  }
  return 0;
}


static R interior_distance(C *w, C c, N period, R pixel_size) {
  if (attractor(w, c, period)) {
    C z = *w;
    C dz = 1.0;
    C dzdz = 0.0;
    C dc = 0.0;
    C dcdz = 0.0;
    for (N j = 0; j < period; ++j) {
      dcdz = 2.0 * (z * dcdz + dz * dc);
      dc = 2.0 * z * dc + 1.0;
      dzdz = 2.0 * (dz * dz + z * dzdz);
      dz = 2.0 * z * dz;
      z = z * z + c;
    }
    return (1.0 - cabs2(dz)) / (cabs(dcdz + dzdz * dc / (1.0 - dz)) * pixel_size↙
        ↳ );
  }
  return -1.0;
}


static void render_recursive(C c, R grid_spacing, N maxiters, N depth) {
  R sqrt2 = sqrt(2);
  C z = 0;
  C dz = 0;
  R mz2 = 1.0/0.0;
  for (N i = 1; i < maxiters; ++i)
  {
    dz = 2 * z * dz + 1;
    z = z * z + c;
    R z2 = cabs2(z);
    if (! (z2 < 65536))
    {
      R de = sqrt(z2) * log(z2) / (cabs(dz) * grid_spacing);
      if (de < sqrt2)
      {
        if (depth > 0)
        {
          render_recursive(c + 1 * 0.5 * grid_spacing + I * 0.5 * grid_spacing, ↙
              ↳ 0.5 * grid_spacing, maxiters, depth - 1);
          render_recursive(c - 1 * 0.5 * grid_spacing + I * 0.5 * grid_spacing, ↙
```

```
            ↪ 0.5 * grid_spacing, maxiters, depth - 1);
          render_recursive(c - 1 * 0.5 * grid_spacing - I * 0.5 * grid_spacing, ↙
            ↪ 0.5 * grid_spacing, maxiters, depth - 1);
          render_recursive(c + 1 * 0.5 * grid_spacing - I * 0.5 * grid_spacing, ↙
            ↪ 0.5 * grid_spacing, maxiters, depth - 1);
        }
      }
      else
      {
        plot_iterates(c, grid_spacing, i);
      }
      break;
    }
    if (z2 < mz2) {
      mz2 = z2;
      C z1 = z;
      R de = interior_distance(&z1, c, i, grid_spacing);
      if (de > 0)
      {
        if (de < sqrt2)
        {
          if (depth > 0)
          {
            render_recursive(c + 1 * 0.5 * grid_spacing + I * 0.5 * grid_spacing ↙
              ↪ , 0.5 * grid_spacing, maxiters, depth - 1);
            render_recursive(c - 1 * 0.5 * grid_spacing + I * 0.5 * grid_spacing ↙
              ↪ , 0.5 * grid_spacing, maxiters, depth - 1);
            render_recursive(c - 1 * 0.5 * grid_spacing - I * 0.5 * grid_spacing ↙
              ↪ , 0.5 * grid_spacing, maxiters, depth - 1);
            render_recursive(c + 1 * 0.5 * grid_spacing - I * 0.5 * grid_spacing ↙
              ↪ , 0.5 * grid_spacing, maxiters, depth - 1);
          }
        }
        break;
      }
    }
  }
}

static void render(N depth) {
  clear();
  N maxiters = 1 << (8 + depth);
  N progress = 0;
  N grid = 256;
  R grid_spacing = 5.0 / grid;
  #pragma omp parallel for schedule(dynamic, 1)
  for (N y = 0; y < grid / 2; ++y) {
    for (N x = 0; x < grid; ++x) {
      C c = grid_spacing * ((x + 0.5 - grid/2.0) + I * (y + 0.5 - grid/2.0));
      render_recursive(c, grid_spacing, maxiters, depth);
    }
    #pragma omp critical
    fprintf(stderr, "%8d\r", ++progress);
  }
  post();
  save();
```

```
}

extern int main(int argc, char **argv) {
  (void) argc;
  (void) argv;
  for (N depth = 0; 1; ++depth)
    render(depth);
  return 0;
}

// END
```

2 buddhabrot/bbcolourizelayers.c

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

typedef int64_t Z;
typedef uint64_t N;


int cmp_N(const void *a, const void *b)
{
  const Z *p = a;
  const Z *q = b;
  Z x = *p;
  Z y = *q;
  return (x > y) - (x < y);
}

int cmp_float(const void *a, const void *b)
{
  const float *p = a;
  const float *q = b;
  float x = *p;
  float y = *q;
  return (x > y) - (x < y);
}



#define LOG2SIZE 12
#define SIZE (1 << LOG2SIZE)
#define LAYERS 30


struct image
{
  N n[LAYERS][SIZE * SIZE];
};
```

```
FILE *image_file = 0;


static struct image *image_map()
{
  image_file = fopen("bb.map", "rb");
  if (! image_file)
  {
    exit(1);
  }
  struct image *img = malloc(sizeof(struct image));
  if (! img)
  {
    exit(1);
  }
  fread(img, sizeof(struct image), 1, image_file);
  return img;
}

static void image_unmap(struct image *img)
{
  fclose(image_file);
  image_file = 0;
  free(img);
}


static double xyz2lab_f(double t)
{
  static const double e = 0.008856;
  static const double k = 903.3;
  if (t > e)
    return cbrt(t);
  else
    return (k * t + 16) / 116;
}
static void xyz2lab(double x, double y, double z, double *l, double *a, double *↙
    ↳ b)
{
  static const double xn = 0.95047;
  static const double yn = 1.00000;
  static const double zn = 1.08883;
  x /= xn;
  y /= yn;
  z /= zn;
  x = xyz2lab_f(x);
  y = xyz2lab_f(y);
  z = xyz2lab_f(z);
  *l = 116 * y - 16;
  *a = 500 * (x - y);
  *b = 200 * (y - z);
}

static double lab2xyz_f1(double t)
{
  static const double e = 0.008856;
```

```
  static const double k = 903.3;
  if (t * t * t > e)
    return t * t * t;
  else
    return (116 * t - 16) / k;
}
static double lab2xyz_f2(double l)
{
  static const double e = 0.008856;
  static const double k = 903.3;
  if (l > k * e)
  {
    double t = (l + 16) / 116;
    return t * t * t;
  }
  else
    return l / k;
}
static void lab2xyz(double l, double a, double b, double *x, double *y, double *z)
{
  static const double xn = 0.95047;
  static const double yn = 1.00000;
  static const double zn = 1.08883;
  double fy = (l + 16) / 116;
  double fz = fy - b / 200;
  double fx = fy + a / 500;
  *x = xn * lab2xyz_f1(fx);
  *y = yn * lab2xyz_f2(l);
  *z = zn * lab2xyz_f1(fz);
}


static double xyz2srgb_f(double c)
{
  if (c < 0.0031308)
    return 12.92 * c;
  else
    return 1.055 * pow(c, 1/2.4) - 0.055;
}
static void xyz2srgb(double x, double y, double z, double *r, double *g, double *b)
{
  static const double m[3][3] =
    { {  3.2406, -1.5372, -0.4986 }
    , { -0.9689,  1.8758,  0.0415 }
    , {  0.0557, -0.2040,  1.0570 }
    };
  *r = xyz2srgb_f(m[0][0] * x + m[0][1] * y + m[0][2] * z);
  *g = xyz2srgb_f(m[1][0] * x + m[1][1] * y + m[1][2] * z);
  *b = xyz2srgb_f(m[2][0] * x + m[2][1] * y + m[2][2] * z);
}


static double srgb2xyz_f(double c)
{
  if (c < 0.04045)
```

```
    return c / 12.92;
  else
    return pow((c + 0.055) / 1.055, 2.4);
}
static void srgb2xyz(double r, double g, double b, double *x, double *y, double *z)
{
  static const double m[3][3] =
    { { 0.4124, 0.3576, 0.1805 }
    , { 0.2126, 0.7152, 0.0722 }
    , { 0.0193, 0.1192, 0.9505 }
    };
  r = srgb2xyz_f(r);
  g = srgb2xyz_f(g);
  b = srgb2xyz_f(b);
  *x = m[0][0] * r + m[0][1] * g + m[0][2] * b;
  *y = m[1][0] * r + m[1][1] * g + m[1][2] * b;
  *z = m[2][0] * r + m[2][1] * g + m[2][2] * b;
}

unsigned char spectrum[LAYERS][3];

int main(int argc, char **argv)
{
  (void) argc;
  (void) argv;
  {
    FILE *sf = fopen("spectrum.ppm", "rb");
    fseek(sf, -LAYERS * 3, SEEK_END);
    fread(spectrum, LAYERS * 3, 1, sf);
    fclose(sf);
  }
  struct image *img = image_map();
  float *raw = calloc(1, sizeof(float) * 3 * SIZE * SIZE);
#if 1
  float *histogram = malloc(sizeof(float) * SIZE * SIZE);
#endif
  unsigned char *ppm = malloc(3 * SIZE * SIZE);
  N total = 0;
  for (int layer = 0; layer < LAYERS; ++layer)
    #pragma omp parallel for reduction(+:total)
    for (Z k = 0; k < SIZE * SIZE; ++k)
      total += img->n[layer][k];
  double scale = SIZE * SIZE * 1.0 / total;
  printf("%lu %e\n", total, scale);
  for (int layer = 0; layer < LAYERS; ++layer)
  {
    char filename[100];
    snprintf(filename, 100, "layer-%02d.pgm", layer);
    double l, a, b;
    { // convert from srgb to lab
      double r, g, bb;
      double x, y, z;
      printf("  %02d\r", layer);
      fflush(stdout);
      r  = spectrum[layer][0]/255.0;
```

```
        g  = spectrum[layer][1]/255.0;
210     bb = spectrum[layer][2]/255.0;
//        printf("r g b: %f %f %f\n", r, g, bb);
        srgb2xyz(r, g, bb, &x, &y, &z);
//        printf("x y z: %f %f %f\n", x, y, z);
        xyz2lab(x, y, z, &l, &a, &b);
215 //        printf("l a b: %f %f %f\n", l, a, b);
//        lab2xyz(l, a, b, &x, &y, &z);
//        printf("x y z: %f %f %f\n", x, y, z);
//        xyz2srgb(x, y, z, &r, &g, &bb);
//        printf("r g b: %f %f %f\n", r, g, bb);
220     l /= LAYERS;
        a /= LAYERS;
        b /= LAYERS;
      }
      #pragma omp parallel for
225   for (Z j = 0; j < SIZE; ++j)
      for (Z i = 0; i < SIZE; ++i)
      {
        Z k = j * SIZE + i;
        N x = img->n[layer][k];
230 #if 0
        double dither = ((((layer * 67 + j) * 236 + i) * 119) & 255) / 256.0;
        double y = 255 * scale * x + dither;
        y = y > 255 ? 255 : y;
        ppm[k] = y;
235 #endif
        raw[3 * k + 0] += scale * x * l;
        raw[3 * k + 1] += scale * x * a;
        raw[3 * k + 2] += scale * x * b;
      }
240 #if 0
      FILE *f = fopen(filename, "wb");
      fprintf(f, "P5\n%d %d\n255\n", SIZE, SIZE);
      fwrite(ppm, SIZE * SIZE, 1, f);
      fclose(f);
245 #endif
    }
    { // clamp
      #pragma omp parallel for
      for (Z k = 0; k < SIZE * SIZE; ++k)
250   {
        double l = raw[3 * k + 0];
        if (l > 100)
        {
          double s = 100 / l;
255       double a = raw[3 * k + 1];
          double b = raw[3 * k + 2];
          l *= s;
          a *= s;
          b *= s;
260       raw[3 * k + 0] = l;
          raw[3 * k + 1] = a;
          raw[3 * k + 2] = b;
        }
      }
265 }
```

```
#if 1
  {
    // auto white balance (make average colour white)
    double l = 0, a = 0, b = 0;
    #pragma omp parallel for reduction(+:l) reduction(+:a) reduction(+:b)
    for (Z k = 0; k < SIZE * SIZE; ++k)
    {
      l += raw[3 * k + 0];
      a += raw[3 * k + 1];
      b += raw[3 * k + 2];
    }
    l /= SIZE * SIZE;
    a /= SIZE * SIZE;
    b /= SIZE * SIZE;
    const double scale = 1.0 / 12 * 100 / l;
    const double shift = 1.0 / 100;
    #pragma omp parallel for
    for (Z k = 0; k < SIZE * SIZE; ++k)
    {
      double t = shift * raw[3 * k + 0];
      raw[3 * k + 1] -= a * t;
      raw[3 * k + 2] -= b * t;
      raw[3 * k + 0] *= scale;
      raw[3 * k + 1] *= scale;
      raw[3 * k + 2] *= scale;
    }
  }
#endif

#if 1
  { // auto-levels
    double quantile_lo = 0.001;
    double quantile_hi = 0.999;
    #pragma omp parallel for
    for (Z k = 0; k < SIZE * SIZE; ++k)
    {
      double l = raw[3 * k + 0];
      double a = raw[3 * k + 1];
      double b = raw[3 * k + 2];
      double x, y, z;
      double r, g, bb;
      lab2xyz(l, a, b, &x, &y, &z);
      xyz2srgb(x, y, z, &r, &g, &bb);
      raw[3 * k + 0] = r;
      raw[3 * k + 1] = g;
      raw[3 * k + 2] = bb;
    }
    for (int c = 0; c < 3; ++c)
    {
      #pragma omp parallel for
      for (Z k = 0; k < SIZE * SIZE; ++k)
        histogram[k] = raw[3 * k + c];
      qsort(histogram, SIZE * SIZE, sizeof(float), cmp_float);
      double lo = histogram[(Z) (SIZE * SIZE * quantile_lo)];
      double hi = histogram[(Z) (SIZE * SIZE * quantile_hi)];
      double range = 1.0 / (hi - lo);
```

```
      if (range > 0) // nan check
      {
        #pragma omp parallel for
        for (Z k = 0; k < SIZE * SIZE; ++k)
        {
          double v = raw[3 * k + c];
          v -= lo;
          v *= range;
          v = v < 0 ? 0 : v;
          v = v > 1 ? 1 : v;
          raw[3 * k + c] = v;
        }
      }
    }
    #pragma omp parallel for
    for (Z k = 0; k < SIZE * SIZE; ++k)
    {
      double r  = raw[3 * k + 0];
      double g  = raw[3 * k + 1];
      double bb = raw[3 * k + 2];
      double x, y, z;
      double l, a, b;
      srgb2xyz(r, g, bb, &x, &y, &z);
      xyz2lab(x, y, z, &l, &a, &b);
      raw[3 * k + 0] = l;
      raw[3 * k + 1] = a;
      raw[3 * k + 2] = b;
    }
  }
#endif


#if 1
  { // increase saturation
    // https://math.stackexchange.com/questions/586424/adjust-saturation-in-cie-↙
        ↳ lab-space
    const double saturation = 1.25;
    const double limit = 0.95;
    #pragma omp parallel for
    for (Z k = 0; k < SIZE * SIZE; ++k)
    {
      double l = raw[3 * k + 0];
      double a = raw[3 * k + 1];
      double b = raw[3 * k + 2];
      double c = hypot(a, b);
      double s = c / hypot(l, c);
      s *= saturation;
      if (s > limit) s = limit;
      double t = s * l / sqrt((a * a + b * b) * (1 - s * s));
      if (t > 0) // nan check
      {
        a *= t;
        b *= t;
      }
      raw[3 * k + 1] = a;
      raw[3 * k + 2] = b;
    }
```

```
  }
#endif

#if 1
  {
    // auto white balance again (make average colour white)
    double l = 0, a = 0, b = 0;
    #pragma omp parallel for reduction(+:l) reduction(+:a) reduction(+:b)
    for (Z k = 0; k < SIZE * SIZE; ++k)
    {
      l += raw[3 * k + 0];
      a += raw[3 * k + 1];
      b += raw[3 * k + 2];
    }
    l /= SIZE * SIZE;
    a /= SIZE * SIZE;
    b /= SIZE * SIZE;
    const double scale = 1.0 / 12 * 100 / l;
    const double shift = 1.0 / 100;
    #pragma omp parallel for
    for (Z k = 0; k < SIZE * SIZE; ++k)
    {
      double t = shift * raw[3 * k + 0];
      raw[3 * k + 1] -= a * t;
      raw[3 * k + 2] -= b * t;
      raw[3 * k + 0] *= scale;
      raw[3 * k + 1] *= scale;
      raw[3 * k + 2] *= scale;
    }
  }
#endif

  { // lab to 8bit srgb
    #pragma omp parallel for
    for (Z j = 0; j < SIZE; ++j)
    for (Z i = 0; i < SIZE; ++i)
    {
      Z k = j * SIZE + i;
#if 1
      double l = raw[3 * k + 0];
      double a = raw[3 * k + 1];
      double b = raw[3 * k + 2];
      double x, y, z, r, g;
      lab2xyz(l, a, b, &x, &y, &z);
      xyz2srgb(x, y, z, &r, &g, &b);
#else
      double r = raw[3 * k + 0];
      double g = raw[3 * k + 1];
      double b = raw[3 * k + 2];
#endif
      double dither;
      dither = ((((0 * 67 + j) * 236 + i) * 119) & 255) / 256.0;
      ppm[3 * k + 0] = fmin(fmax(r * 255 + dither, 0), 255);
      dither = ((((1 * 67 + j) * 236 + i) * 119) & 255) / 256.0;
      ppm[3 * k + 1] = fmin(fmax(g * 255 + dither, 0), 255);
      dither = ((((2 * 67 + j) * 236 + i) * 119) & 255) / 256.0;
      ppm[3 * k + 2] = fmin(fmax(b * 255 + dither, 0), 255);
```

```
    }
  }
  FILE *f = fopen("colourized.ppm", "wb");
  fprintf(f, "P6\n%d %d\n255\n", SIZE, SIZE);
  fwrite(ppm, 3 * SIZE * SIZE, 1, f);
  fclose(f);
  free(ppm);
  image_unmap(img);
}
```

3 buddhabrot/bbrender.c

```
// gcc -std=c99 -Wall -pedantic -Wextra -O3 -fopenmp -o bbrender bbrender.c `↙
    ↪ PKG_CONFIG_PATH=${HOME}/opt/lib/pkgconfig pkg-config --cflags --libs ↙
    ↪ mandelbrot-numerics` -lm
// ./bbrender | pnmsplit - bbrender-%d.pgm

#include <complex.h>
#include <math.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

#include <mandelbrot-numerics.h>

typedef unsigned char B;
typedef float F;
typedef uint64_t N;
typedef int64_t Z;
typedef double R;
typedef double _Complex C;

#define unlikely(x) __builtin_expect(x, 0)
#undef BB_BOUNDS_CHECKS

struct partial {
  C z;
  Z p;
};


struct compute {
  Z tag, bias;
  struct partial *partials;
  Z npartials, np, n;
  R er2, mz2;
  C c, z;
};

static struct compute *compute_new(Z npartials) {
  struct compute *px = malloc(sizeof(*px));
  if (! px) {
    return 0;
  }
  if (npartials > 0) {
    px->partials = malloc(npartials * sizeof(*(px->partials)));
```

```
    px->npartials = npartials;
    if (! px->partials) {
      free(px);
      return 0;
    }
  } else {
    px->partials = 0;
    px->npartials = 0;
  }
  px->tag = 0;
  return px;
}


static void compute_delete(struct compute *px) {
  if (px) {
    if (px->partials) {
      free(px->partials);
    }
    free(px);
  }
}


static void compute_init(struct compute *px, Z bias, C c) {
  if (! px) { return; }
  px->tag = 0;
  px->bias = bias;
  px->mz2 = 1.0 / 0.0;
  px->c = c;
  px->z = 0;
  px->n = 0;
  px->np = 0;
}

static bool is_interior(C c, C z, Z p, Z steps)
{
  C z00 = 0;
  if (m_failed != m_d_attractor(&z00, z, c, p, steps)) {
    C z0 = z00;
    C dz0 = 1;
    for (Z j = 0; j < p; ++j)
    {
      dz0 = 2.0 * z0 * dz0;
      z0 = z0 * z0 + c;
    }
    return creal(dz0)*creal(dz0) + cimag(dz0)*cimag(dz0) <= 1;
  }
  return false;
}

extern bool compute_step(struct compute *px, Z steps)
{
  static const double er2 = 4;
  if (! px) {
    return false;
  }
```

```
  if (px->tag != 0) {
    return true;
  }
  C c = px->c;
  C z = px->z;
  R mz2 = px->mz2;
  Z n = px->n;
  R cx = creal(c);
  R cy = cimag(c);
  R zx = creal(z);
  R zy = cimag(z);
  R zx2 = zx * zx;
  R zy2 = zy * zy;
  R zxy = zx * zy;
  if (px->bias < 0)
  {
    for (Z i = 1; i <= steps; ++i)
    {
      zx = zx2 - zy2 + cx;
      zy = zxy + zxy + cy;
      zx2 = zx * zx;
      zy2 = zy * zy;
      zxy = zx * zy;
      R z2 = zx2 + zy2;
      if (unlikely(z2 < mz2))
      {
        mz2 = z2;
        if (is_interior(c, zx + I * zy, n + i, 16))
        {
          px->tag = -1;
          return true;
        }
      }
      if (unlikely(z2 >= er2))
      {
        px->tag = 1;
        px->n = n + i;
        return true;
      }
    }
    px->tag = 0;
    px->z = zx + I * zy;
    px->mz2 = mz2;
    px->n = n + steps;
    return false;
  }
  else
  {
    for (Z i = 1; i <= steps; ++i)
    {
      zx = zx2 - zy2 + cx;
      zy = zxy + zxy + cy;
      zx2 = zx * zx;
      zy2 = zy * zy;
      zxy = zx * zy;
      R z2 = zx2 + zy2;
      if (unlikely(z2 < mz2))
```

```
      {
        mz2 = z2;
        if (px->partials && px->np < px->npartials)
        {
          px->partials[px->np].z = z;
          px->partials[px->np].p = n + i;
          px->np = px->np + 1;
        }
      }
      if (unlikely(z2 >= er2))
      {
        px->tag = 1;
        px->n = n + i;
        return true;
      }
    }
    for (Z i = 0; i < px->np; ++i)
    {
      z = px->partials[i].z;
      Z p = px->partials[i].p;
      if (is_interior(c, z, p, 16))
      {
        px->tag = -1;
        px->z = z;
        px->n = p;
        return true;
      }
    }
    px->tag = 0;
    px->z = zx + I * zy;
    px->mz2 = mz2;
    px->n = n + steps;
    return false;
  }
  return false;
}

#define LEVELS 14

struct image
{
  N *n[LEVELS];
  B *b[LEVELS];
  F *f[2][LEVELS];
};

static struct image *image_new()
{
  struct image *img = calloc(1, sizeof(*img));
  for (Z level = 0; level < LEVELS; ++level)
  {
    Z grid = 1 << level;
    Z bytes = grid * grid;
    img->n[level] = calloc(1, sizeof(N) * bytes);
```

```
215       img->b[level] = calloc(1, bytes);
          img->f[0][level] = calloc(1, sizeof(F) * bytes);
          img->f[1][level] = calloc(1, sizeof(F) * bytes);
        }
        return img;
220   }


      static void image_delete(struct image *img)
      {
225     for (Z level = 0; level < LEVELS; ++level)
        {
          free(img->n[level]);
          free(img->b[level]);
          free(img->f[0][level]);
230       free(img->f[1][level]);
        }
        free(img);
      }

235
      static inline void image_plot(struct image *img, R zx, R zy)
      {
        // flipped along main diagonal
      #ifdef BB_BOUNDS_CHECKS
240     Z y = floor((1 << (LEVELS-1)) * (zx + 2.0) / 4.0);
        Z x = floor((1 << (LEVELS-1)) * (zy + 2.0) / 4.0);
        if (0 <= x && x < (1 << (LEVELS-1)) && 0 <= y && y < (1 << (LEVELS-1)))
        {
      #else
245     Z y = (1 << (LEVELS-1)) * (zx + 2.0) / 4.0;
        Z x = (1 << (LEVELS-1)) * (zy + 2.0) / 4.0;
      #endif
          Z k = (y << (LEVELS - 1)) + x;
          N *p = img->n[LEVELS - 1] + k;
250       #pragma omp atomic
          *p += 1;
      #ifdef BB_BOUNDS_CHECKS
        }
      #endif
255   }


      static void image_post(struct image *img)
      {
260     for (Z src = LEVELS - 1; src > 0; --src)
        {
          Z dst = src - 1;
          N *srcp = img->n[src];
          N *dstp = img->n[dst];
265       #pragma omp parallel for
          for (Z y = 0; y < (1 << dst); ++y)
            for (Z x = 0; x < (1 << dst); ++x)
            {
              N s = 0;
270           for (Z dy = 0; dy < 2; ++dy)
                for (Z dx = 0; dx < 2; ++dx)
```

```
            {
              Z k = (((y << 1) + dy) << src) + ((x << 1) + dx);
              s += srcp[k];
            }
          Z k = (y << dst) + x;
          dstp[k] = s;
        }
    }
    N total = img->n[0][0];
    for (Z level = 0; level < LEVELS; ++level)
    {
      Z pixels = 1 << level;
      pixels *= pixels;
      R average = total / (R) pixels;
      R norm = 1.0 / average;
      R scale = 255.0 / (16.0 * average);
      N *n = img->n[level];
      F *f = img->f[0][level];
      F *g = img->f[1][level];
      B *b = img->b[level];
      #pragma omp parallel for
      for (Z k = 0; k < pixels; ++k)
      {
        Z x = k & ((1 << level) - 1);
        Z y = k >> level;
        R dither = (((y * 237 + x) * 119) & 255) / 256.0;
        g[k] = f[k];
        f[k] = norm * n[k];
        b[k] = floor(fmin(fmax(scale * n[k] + dither, 0.0), 255.0));
      }
    }
  }


  static R image_rms_diff(struct image *img, Z level)
  {
    Z pixels = 1 << level;
    pixels *= pixels;
    F *f = img->f[0][level];
    F *g = img->f[1][level];
    R sum = 0;
    #pragma omp parallel for reduction(+:sum)
    for (Z k = 0; k < pixels; ++k)
    {
      R d = f[k] - g[k];
      d *= d;
      sum = sum + d;
    }
    R rms = sqrt(sum / pixels);
    return rms;
  }


  static void image_save(struct image *img, Z level, Z depth)
  {
    Z pixels = 1 << level;
    pixels *= pixels;
```

```
  char filename[100];
  snprintf(filename, 100, "bb-%02d-%02d.pgm", (int) depth, (int) level);
  FILE *out = fopen(filename, "wb");
  fprintf(out, "P5\n%d %d\n255\n", 1 << level, 1 << level);
  fwrite(img->b[level], pixels, 1, out);
  fflush(out);
  fclose(out);
  snprintf(filename, 100, "bb-%02d-%02d.u64", (int) depth, (int) level);
  out = fopen(filename, "wb");
  fwrite(img->n[level], sizeof(N) * pixels, 1, out);
  fflush(out);
  fclose(out);
}


static void render(struct image *img, Z maxiters, Z grid, C mul, C add)
{
  #pragma omp parallel for schedule(dynamic, 1)
  for (Z j = 0; j < grid; ++j)
  {
    struct compute *px = compute_new(maxiters);
    for (Z i = 0; i < grid; ++i)
    {
      C c = mul * (i + I * j) + add;
      compute_init(px, px->tag, c);
      compute_step(px, maxiters);
      if (px->tag > 0)
      {
        R zx = 0;
        R zy = 0;
        R cx = creal(c);
        R cy = cimag(c);
        Z count = px->n - 1;
        R zx2 = zx * zx;
        R zy2 = zy * zy;
        R zxy = zx * zy;
        for (Z n = 0; n < count; ++n)
        {
          zx = zx2 - zy2 + cx;
          zy = zxy + zxy + cy;
          zx2 = zx * zx;
          zy2 = zy * zy;
          zxy = zx * zy;
          image_plot(img, zx, zy);
        }
      }
    }
    compute_delete(px);
  }
}


extern int main(int argc, char **argv) {
  (void) argc;
  (void) argv;
  srand(0xbeefcafe);
  static const R threshold = 0.05;
```

```
  for (Z level = LEVELS - 1; level < LEVELS; ++level)
  {
    for (Z depth = 20; depth <= 20; ++depth)
    {
      Z maxiters = 1 << depth;
      Z pixels = 0;
      struct image *img = image_new();
      for (Z pass = 0; 1; ++pass)
      {
        for (Z count = 0; count < 1 << pass; ++count)
        {
          fprintf(stderr, "\r %ld, %ld : %ld : %ld : %ld ", level, depth, pass, ↙
              ↳ ((Z)1) << pass, count);
          fflush(stderr);
          // randomize grid
          Z n = exp2(10 + (rand() / (R) RAND_MAX));
          n += (n & 1); // ensure even
          R l = 2.0 / (n/2 - 1) + (2.0/(n/2-3) - 2.0/(n/2-1)) * (rand() / (R) ↙
              ↳ RAND_MAX);
          l *= 3.6;
          R x = (rand() / (R) RAND_MAX);
          R y = (rand() / (R) RAND_MAX);
          R t = 2 * 3.141592653589793 * (rand() / (R) RAND_MAX);
          C mul = l * cexp(I * t);
          C add = - mul * ((n/2 + x) + I * ((n/2) + y));
          // calculate
          render(img, maxiters, n, mul, add);
          // output
          pixels += n * n;
        }
        image_post(img);
        fprintf(stderr, "\r saving... ");
        fflush(stderr);
        image_save(img, level, depth);
        fprintf(stderr, "saved! ");
        fflush(stderr);
        if (pass)
        {
          R rms = image_rms_diff(img, level);
          if (rms < threshold)
          {
            fprintf(stderr, "\n");
            fflush(stderr);
            printf("%ld %ld %ld %.18e %ld %lu\n", level, depth, pass, rms, ↙
                ↳ pixels, img->n[0][0]);
            fflush(stdout);
            break;
          }
        }
      }
      image_delete(img);
    }
  }
  return 0;
}
```

```
440 // END
```

4 buddhabrot/bbrenderlayers.c

```
// gcc -std=c99 -Wall -pedantic -Wextra -O3 -fopenmp -o bbrenderlayers ↙
    ↪ bbrenderlayers.c `PKG_CONFIG_PATH=${HOME}/opt/lib/pkgconfig pkg-config --↙
    ↪ cflags --libs mandelbrot-numerics` -lm
// dd if=/dev/zero of=bb.map bs=$((1024 * 1024)) count=1024
// ./bbrenderlayers

#include <complex.h>
#include <math.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <signal.h>
#include <time.h>
#include <mandelbrot-numerics.h>


typedef unsigned char B;
typedef float F;
typedef uint64_t N;
typedef int64_t Z;
typedef double R;
typedef double _Complex C;


#define unlikely(x) __builtin_expect(x, 0)


Z ilog2(Z n)
{
  Z l = 0;
  while (n > 0)
  {
    n >>= 1;
    l += 1;
  }
  return l;
}


struct partial {
  C z;
  Z p;
};


struct compute {
  Z tag, bias;
  struct partial *partials;
  Z npartials, np, n;
  R er2, mz2;
  C c, z;
};
```

```
static struct compute *compute_new(Z npartials) {
  struct compute *px = malloc(sizeof(*px));
  if (! px) {
    return 0;
  }
  if (npartials > 0) {
    px->partials = malloc(npartials * sizeof(*(px->partials)));
    px->npartials = npartials;
    if (! px->partials) {
      free(px);
      return 0;
    }
  } else {
    px->partials = 0;
    px->npartials = 0;
  }
  px->tag = 0;
  return px;
}


static void compute_delete(struct compute *px) {
  if (px) {
    if (px->partials) {
      free(px->partials);
    }
    free(px);
  }
}


static void compute_init(struct compute *px, Z bias, C c) {
  if (! px) { return; }
  px->tag = 0;
  px->bias = bias;
  px->mz2 = 1.0 / 0.0;
  px->c = c;
  px->z = 0;
  px->n = 0;
  px->np = 0;
}


static bool is_interior(C c, C z, Z p, Z steps)
{
  C z00 = 0;
  if (m_failed != m_d_attractor(&z00, z, c, p, steps)) {
    C z0 = z00;
    C dz0 = 1;
    for (Z j = 0; j < p; ++j)
    {
      dz0 = 2.0 * z0 * dz0;
      z0 = z0 * z0 + c;
    }
    return creal(dz0)*creal(dz0) + cimag(dz0)*cimag(dz0) <= 1;
  }
```

```
  return false;
}

static bool compute_step(struct compute *px, Z steps)
{
  static const double er2 = 4;
  if (! px) {
    return false;
  }
  if (px->tag != 0) {
    return true;
  }
  C c = px->c;
  C z = px->z;
  R mz2 = px->mz2;
  Z n = px->n;
  R cx = creal(c);
  R cy = cimag(c);
  R zx = creal(z);
  R zy = cimag(z);
  R zx2 = zx * zx;
  R zy2 = zy * zy;
  R zxy = zx * zy;
  if (px->bias < 0)
  {
    for (Z i = 1; i <= steps; ++i)
    {
      zx = zx2 - zy2 + cx;
      zy = zxy + zxy + cy;
      zx2 = zx * zx;
      zy2 = zy * zy;
      zxy = zx * zy;
      R z2 = zx2 + zy2;
      if (unlikely(z2 < mz2))
      {
        mz2 = z2;
        if (is_interior(c, zx + I * zy, n + i, 16))
        {
          px->tag = -1;
          return true;
        }
      }
      if (unlikely(z2 >= er2))
      {
        px->tag = 1;
        px->n = n + i;
        return true;
      }
    }
    px->tag = 0;
    px->z = zx + I * zy;
    px->mz2 = mz2;
    px->n = n + steps;
    return false;
  }
  else
```

```
  {
    for (Z i = 1; i <= steps; ++i)
    {
      zx = zx2 - zy2 + cx;
      zy = zxy + zxy + cy;
      zx2 = zx * zx;
      zy2 = zy * zy;
      zxy = zx * zy;
      R z2 = zx2 + zy2;
      if (unlikely(z2 < mz2))
      {
        mz2 = z2;
        if (px->partials && px->np < px->npartials)
        {
          px->partials[px->np].z = z;
          px->partials[px->np].p = n + i;
          px->np = px->np + 1;
        }
      }
      if (unlikely(z2 >= er2))
      {
        px->tag = 1;
        px->n = n + i;
        return true;
      }
    }
    for (Z i = 0; i < px->np; ++i)
    {
      z = px->partials[i].z;
      Z p = px->partials[i].p;
      if (is_interior(c, z, p, 16))
      {
        px->tag = -1;
        px->z = z;
        px->n = p;
        return true;
      }
    }
    px->tag = 0;
    px->z = zx + I * zy;
    px->mz2 = mz2;
    px->n = n + steps;
    return false;
  }
  return false;
}


#define LOG2SIZE 12
#define SIZE (1 << LOG2SIZE)
#define LAYERS 30


struct image
{
  N n[LAYERS][SIZE * SIZE];
};
```

```
225 FILE *image_file = 0;


    static struct image *image_map()
    {
230   image_file = fopen("bb.map", "r+b");
      if (! image_file)
      {
        exit(1);
      }
235   struct image *img = malloc(sizeof(struct image));
      if (! img)
      {
        exit(1);
      }
240   fread(img, sizeof(struct image), 1, image_file);
      return img;
    }


245 static void image_sync(struct image *img)
    {
      rewind(image_file);
      fwrite(img, sizeof(struct image), 1, image_file);
    }
250

    static void image_unmap(struct image *img)
    {
      image_sync(img);
255   fclose(image_file);
      image_file = 0;
      free(img);
    }

260
    static inline void image_plot_with_bounds_checks(struct image *img, Z layer, R ↙
        ↳ zx, R zy)
    {
      // flipped along main diagonal
      Z y = floor(SIZE * (zx + 2.0) / 4.0);
265   Z x = floor(SIZE * (zy + 2.0) / 4.0);
      if (0 <= x && x < SIZE && 0 <= y && y < SIZE)
      {
        Z k = (y << LOG2SIZE) + x;
        N *p = img->n[layer] + k;
270     #pragma omp atomic
        *p += 1;
      }
    }

275
    static inline void image_plot_without_bounds_checks(struct image *img, Z layer, ↙
        ↳ R zx, R zy)
    {
```

```
  // flipped along main diagonal
  Z y = SIZE * (zx + 2.0) / 4.0;
  Z x = SIZE * (zy + 2.0) / 4.0;
  Z k = (y << LOG2SIZE) + x;
  N *p = img->n[layer] + k;
  #pragma omp atomic
  *p += 1;
}


static Z render(struct image *img, Z maxiters, Z grid, C mul, C add)
{
  Z maxpartials = 65536;
  Z total = 0;
  #pragma omp parallel for schedule(dynamic, 1) reduction(+:total)
  for (Z j = 0; j < grid; ++j)
  {
    struct compute *px = compute_new(maxpartials);
    Z stotal = 0;
    for (Z i = 0; i < grid; ++i)
    {
      C c = mul * (i + I * j) + add;
      compute_init(px, px->tag, c);
      for (Z iters = 256; iters < maxiters; iters <<= 1)
      {
        compute_step(px, iters);
        if (px->tag > 0)
        {
          R zx = 0;
          R zy = 0;
          R cx = creal(c);
          R cy = cimag(c);
          Z count = px->n - 1;
          Z layer = ilog2(count);
          if (! (0 <= layer && layer < LAYERS))
            continue;
          R zx2 = zx * zx;
          R zy2 = zy * zy;
          R zxy = zx * zy;
          for (Z n = 0; n < count; ++n)
          {
            zx = zx2 - zy2 + cx;
            zy = zxy + zxy + cy;
            zx2 = zx * zx;
            zy2 = zy * zy;
            zxy = zx * zy;
            image_plot_without_bounds_checks(img, layer, zx, zy);
          }
          while (zx2 + zy2 < 16.0)
          {
            zx = zx2 - zy2 + cx;
            zy = zxy + zxy + cy;
            zx2 = zx * zx;
            zy2 = zy * zy;
            zxy = zx * zy;
            image_plot_with_bounds_checks(img, layer, zx, zy);
          }
```

```
335             stotal += count;
              }
              if (px->tag < 0)
                break;
            }
340       }
          compute_delete(px);
          total = total + stotal;
        }
        return total;
345   }


      static volatile bool running = true;

350

      static void handler(int sig)
      {
        (void) sig;
        running = false;
355   }


      extern int main(int argc, char **argv) {
        (void) argc;
360     (void) argv;
        Z seed = time(0);
        fprintf(stdout, "starting with seed %016lx\n", seed);
        fflush(stdout);
        srand(seed);
365     Z depth = 30;
        Z maxiters = 1LL << depth;
        Z pixels = 0;
        Z total = 0;
        struct image *img = image_map();
370     time_t last = time(0);
        signal(SIGINT, handler);
        signal(SIGTERM, handler);
        while (running)
        {
375       // randomize grid
          Z n = exp2(14 + (rand() / (R) RAND_MAX));
          n += (n & 1); // ensure even
          R l = 2.0 / (n/2 - 1) + (2.0/(n/2-3) - 2.0/(n/2-1)) * (rand() / (R) RAND_MAX↙
              ↳ );
          l *= sqrt(2);
380       R x = (rand() / (R) RAND_MAX);
          R y = (rand() / (R) RAND_MAX);
          R t = 2 * 3.141592653589793 * (rand() / (R) RAND_MAX);
          C mul = l * cexp(I * t);
          C add = - mul * ((n/2 + x) + I * ((n/2) + y));
385       // calculate
          total += render(img, maxiters, n, mul, add);
          pixels += n * n;
          // output
          fprintf(stdout, "%18ld / %18ld = %.18f\n", total, pixels, total / (double) ↙
              ↳ pixels);
```

```
      fflush(stdout);
      time_t now = time(0);
      if (now - last >= 60 * 60) // 1hr
      {
        fprintf(stdout, "syncing...\n");
        fflush(stdout);
        image_sync(img);
        last = time(0);
        fprintf(stdout, "...synced\n");
        fflush(stdout);
      }
    }
    fprintf(stdout, "exiting\n");
    fflush(stdout);
    image_unmap(img);
    return 0;
}


// END
```

5 buddhabrot/bound-2.c

```
// gcc -std=c99 -Wall -pedantic -Wextra -O3 -fopenmp -o bound-2 bound-2.c `↙
    ↪ PKG_CONFIG_PATH=${HOME}/opt/lib/pkgconfig pkg-config --cflags --libs ↙
    ↪ mandelbrot-graphics` -lm
// ./bound-2 | tee bound-2.dat

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

#include <mandelbrot-graphics.h>

typedef uint64_t N;
typedef int64_t Z;
typedef double R;
typedef double _Complex C;

static R render(N maxiters, N grid, C mul, C add) {
  R s = 0;
  R side = cabs(mul) * grid;
  R area = side * side;
  R count = grid * grid;
  R scale = area / count;
  #pragma omp parallel for schedule(dynamic, 1) reduction(+:s)
  for (N j = 0; j < grid; ++j) {
    R ls = 0;
    m_d_compute *px = m_d_compute_alloc(maxiters);
    for (N i = 0; i < grid; ++i) {
      C c = mul * (i + I * j) + add;
      m_d_compute_init(px, m_d_compute_get_tag(px), 2, c, 0);
      m_d_compute_step(px, maxiters);
      if (m_d_compute_get_tag(px) == m_exterior)
      {
        Z n = m_d_compute_get_n(px) - 1;
```

```
        R a = n * scale;
        ls += a;
      }
    }
    m_d_compute_free(px);
    s = s + ls;
  }
  printf("%.18f %d %d %.18f %.18f %.18f %.18f\n", s, (int) maxiters, (int) grid, creal(mul), cimag(mul), creal(add), cimag(add));
  fflush(stdout);
  return s;
}

extern int main(int argc, char **argv) {
  (void) argc;
  (void) argv;
  srand(0xbeefcafe);
  printf("# total maxiters grid mulre mulim addre addim\n");
  for (N depth = 8; depth < 17; ++depth)
  {
    N count = 1024 >> (depth - 8);
    for (N i = 0; i < count; ++i)
    {
      N n = exp2(depth + (rand() / (R) RAND_MAX));
      n += (n & 1); // ensure even
      R l = 2.0 / (n/2 - 1) + (2.0/(n/2-3) - 2.0/(n/2-1)) * (rand() / (R) RAND_MAX);
      R x = (rand() / (R) RAND_MAX);
      R y = (rand() / (R) RAND_MAX);
      R t = 2 * 3.141592653589793 * (rand() / (R) RAND_MAX);
/*
      mul * ((n/2 + x) + I * (n/2 + y)) + add = 0;
      abs mul = l
      arg mul = t
      add = - mul(...)
*/
      C mul = l * cexp(I * t);
      C add = - mul * ((n/2 + x) + I * ((n/2) + y));
      render(256 * n, n, mul, add);
    }
  }
  return 0;
}

// END
```

6 buddhabrot/bound-2.gnuplot

```
set terminal pngcairo enhanced font "LMSans10,18" size 1920,1080
set output 'bound-2d.png'
set title 'Is the Buddhabrot well-defined? More data needed...'
set xlabel 'Grid size (pixels per edge)'
set ylabel 'Sum (n A_n) (iteration count times area)'
set key bottom right
set grid
```

```
set log x 2
set xtics 2
set fit errorvariables
fit A+B*log(x) 'bound-2.dat' u 3:1:(1./$3) via A,B
fit C+D/log(x) 'bound-2.dat' u 3:1:(1./$3) via C,D
plot [256:131072][40:100]\
  'x.dat' u 1:((A-A_err)+(B-B_err)*log($1)):((A+A_err)+(B+B_err)*log($1)) w ↙
     ↳ filledcurves lc rgb '#80ff0000' t 'fit error bounds', \
  'x.dat' u 1:((C-C_err)+(D-D_err)/log($1)):((C+C_err)+(D+D_err)/log($1)) w ↙
     ↳ filledcurves lc rgb '#8000ff00' t 'fit error bounds', \
  'bound-2.dat' u 3:1 w p lt 6 lc -1 t 'computed grids', \
  A+B*log(x) w l lc rgb '#ff0000' t 'fit (A + B * log x)', \
  C+D/log(x) w l lc rgb '#00ff00' t 'fit (C + D / log x)', \
  C w l lc rgb '#408040' t 'fit asymptote (C)'
```

7 buddhabrot/bound.c

```
// gcc -std=c99 -Wall -pedantic -Wextra -O3 -fopenmp -o bound bound.c `↙
    ↳ PKG_CONFIG_PATH=${HOME}/opt/lib/pkgconfig pkg-config --cflags --libs ↙
    ↳ mandelbrot-graphics`
// ./bound | tee bound.dat

#include <complex.h>
#include <math.h>
#include <stdio.h>

#include <mandelbrot-graphics.h>

typedef uint64_t N;
typedef int64_t Z;
typedef double R;
typedef double _Complex C;

static void render(N depth) {
  N maxiters = ((N)1) << depth;
  N grid = ((N)1) << depth;
  R s = 0;
  #pragma omp parallel for schedule(dynamic, 1) reduction(+:s)
  for (N j = 0; j < grid/2; ++j) {
    R ls = 0;
    m_d_compute *px = m_d_compute_alloc(maxiters);
    for (N i = 0; i < grid; ++i) {
      C c = (-2.5 - 2.5 * I) + 5.0 * ((i + 0.5) / grid + I * (j + 0.5) / grid);
      m_d_compute_init(px, m_d_compute_get_tag(px), 600, c, 0);
      m_d_compute_step(px, maxiters);
      if (m_d_compute_get_tag(px) == m_exterior)
      {
        R d = m_d_compute_get_de(px);
        Z n = m_d_compute_get_n(px);
        R a = n * d * d / (grid/2.0 * grid);
        ls += a;
      }
    }
    m_d_compute_free(px);
    s = s + ls;
  }
  printf("%d %.18f\n", (int)depth, s);
```

```
  fflush(stdout);
}


extern int main(int argc, char **argv) {
  (void) argc;
  (void) argv;
  for (N depth = 8; 1; ++depth)
    render(depth);
  return 0;
}

// END
```

8 buddhabrot/bs.c

```
#include <math.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <string.h>

#define MAXITERS (1LL << 30)

#define l 12
unsigned char state[1<<l][1<<l];
unsigned char priority[1<<l][1<<l];
#define escaped 255
#define unescaped 0
#define unknown 128

int main(int argc, char **argv)
{
  memset(state, unknown, (1 << l) * (1 << l));
  memset(priority, 0, (1 << l) * (1 << l));
  for (int i = 0; i < 1 << l; ++i)
  {
    priority[0][i] = 1;
    priority[i][0] = 1;
    priority[(1<<l)-1][i] = 1;
    priority[i][(1<<l)-1] = 1;
  }
  int passes = 0;
  bool have_some_queued;
  do
  {
    have_some_queued = false;
    fprintf(stderr, "%d\n", passes++);
    #pragma omp parallel for schedule(dynamic, 1)
    for (int i = 0; i < 1 << l; ++i)
      for (int j = 0; j < 1 << l; ++j)
      {
        if (priority[i][j] && state[i][j] != escaped && state[i][j] != unescaped↙
            ↳ )
        {
          have_some_queued = true;
```

```
40            // flipped along main diagonal
              double cy = 4.0 * (i + 0.5) / (1 << l) - 2.0;
              double cx = 4.0 * (j + 0.5) / (1 << l) - 2.0;
              double zx = 0;
              double zy = 0;
45            double zx2 = zx * zx;
              double zy2 = zy * zy;
              double zxy = fabs(zx * zy);
              int64_t count;
              int e = 0;
50            for (count = 1; count < MAXITERS; ++count)
              {
                zx = zx2 - zy2 + cx;
                zy = zxy + zxy + cy;
                zx2 = zx * zx;
55              zy2 = zy * zy;
                zxy = fabs(zx * zy);
                if ((zx2 + zy2 > 4))
                {
                  e = 1;
60                break;
                }
              }
              state[i][j] = e ? escaped : unescaped;
              if (e)
65            {
                for (int ii = i - 1; ii  <= i + 1; ++ii)
                {
                  for (int jj = j - 1; jj <= j + 1; ++jj)
                  {
70                  if (0 <= ii && ii < 1 << l && 0 <= jj && jj < 1 << l)
                    {
//                      #pragma omp atomic
                      priority[ii][jj] += 1;
                    }
75                }
                }
              }
            }
          }
80    } while (have_some_queued);
      fprintf(stdout, "P5\n%d %d\n255\n", 1 << l, 1 << l);
      fwrite(&state[0][0], (1 << l) * (1 << l), 1, stdout);
      return 0;
    }
```

9 buddhabrot/bscolourizelayers.c

```
    #include <math.h>
    #include <stdint.h>
    #include <stdio.h>
    #include <stdlib.h>
5   #include <string.h>

    typedef int64_t Z;
    typedef uint64_t N;
```

```
10
int cmp_N(const void *a, const void *b)
{
  const Z *p = a;
  const Z *q = b;
15  Z x = *p;
  Z y = *q;
  return (x > y) - (x < y);
}

20 int cmp_float(const void *a, const void *b)
{
  const float *p = a;
  const float *q = b;
  float x = *p;
25  float y = *q;
  return (x > y) - (x < y);
}


30
#define LOG2SIZE 12
#define SIZE (1 << LOG2SIZE)
#define LAYERS 30

35
struct image
{
  N n[LAYERS][SIZE * SIZE];
};
40

FILE *image_file = 0;


45 static struct image *image_map()
{
  image_file = fopen("bb.map", "rb");
  if (! image_file)
  {
50  exit(1);
  }
  struct image *img = malloc(sizeof(struct image));
  if (! img)
  {
55  exit(1);
  }
  fread(img, sizeof(struct image), 1, image_file);
  return img;
}
60

static void image_unmap(struct image *img)
{
  fclose(image_file);
65  image_file = 0;
  free(img);
```

```
}


static double xyz2lab_f(double t)
{
  static const double e = 0.008856;
  static const double k = 903.3;
  if (t > e)
    return cbrt(t);
  else
    return (k * t + 16) / 116;
}
static void xyz2lab(double x, double y, double z, double *l, double *a, double *↙
    ↳ b)
{
  static const double xn = 0.95047;
  static const double yn = 1.00000;
  static const double zn = 1.08883;
  x /= xn;
  y /= yn;
  z /= zn;
  x = xyz2lab_f(x);
  y = xyz2lab_f(y);
  z = xyz2lab_f(z);
  *l = 116 * y - 16;
  *a = 500 * (x - y);
  *b = 200 * (y - z);
}

static double lab2xyz_f1(double t)
{
  static const double e = 0.008856;
  static const double k = 903.3;
  if (t * t * t > e)
    return t * t * t;
  else
    return (116 * t - 16) / k;
}
static double lab2xyz_f2(double l)
{
  static const double e = 0.008856;
  static const double k = 903.3;
  if (l > k * e)
  {
    double t = (l + 16) / 116;
    return t * t * t;
  }
  else
    return l / k;
}
static void lab2xyz(double l, double a, double b, double *x, double *y, double *↙
    ↳ z)
{
  static const double xn = 0.95047;
  static const double yn = 1.00000;
  static const double zn = 1.08883;
  double fy = (l + 16) / 116;
```

```
  double fz = fy - b / 200;
  double fx = fy + a / 500;
  *x = xn * lab2xyz_f1(fx);
  *y = yn * lab2xyz_f2(l);
  *z = zn * lab2xyz_f1(fz);
}


static double xyz2srgb_f(double c)
{
  if (c < 0.0031308)
    return 12.92 * c;
  else
    return 1.055 * pow(c, 1/2.4) - 0.055;
}
static void xyz2srgb(double x, double y, double z, double *r, double *g, double ↙
    ↳ *b)
{
  static const double m[3][3] =
    { {  3.2406, -1.5372, -0.4986 }
    , { -0.9689,  1.8758,  0.0415 }
    , {  0.0557, -0.2040,  1.0570 }
    };
  *r = xyz2srgb_f(m[0][0] * x + m[0][1] * y + m[0][2] * z);
  *g = xyz2srgb_f(m[1][0] * x + m[1][1] * y + m[1][2] * z);
  *b = xyz2srgb_f(m[2][0] * x + m[2][1] * y + m[2][2] * z);
}


static double srgb2xyz_f(double c)
{
  if (c < 0.04045)
    return c / 12.92;
  else
    return pow((c + 0.055) / 1.055, 2.4);
}
static void srgb2xyz(double r, double g, double b, double *x, double *y, double ↙
    ↳ *z)
{
  static const double m[3][3] =
    { { 0.4124, 0.3576, 0.1805 }
    , { 0.2126, 0.7152, 0.0722 }
    , { 0.0193, 0.1192, 0.9505 }
    };
  r = srgb2xyz_f(r);
  g = srgb2xyz_f(g);
  b = srgb2xyz_f(b);
  *x = m[0][0] * r + m[0][1] * g + m[0][2] * b;
  *y = m[1][0] * r + m[1][1] * g + m[1][2] * b;
  *z = m[2][0] * r + m[2][1] * g + m[2][2] * b;
}


unsigned char spectrum[30][3];

int main(int argc, char **argv)
{
```

```
  (void) argc;
  (void) argv;
  {
    FILE *sf = fopen("spectrum.ppm", "rb");
    fseek(sf, -LAYERS * 3, SEEK_END);
    fread(spectrum, LAYERS * 3, 1, sf);
    fclose(sf);
  }
  struct image *img = image_map();
  float *raw = calloc(1, sizeof(float) * 3 * SIZE * SIZE);
#if 1
  float *histogram = malloc(sizeof(float) * SIZE * SIZE);
#endif
  unsigned char *ppm = malloc(3 * SIZE * SIZE);
  N total = 0;
  for (int layer = 0; layer < LAYERS; ++layer)
    #pragma omp parallel for reduction(+:total)
    for (Z k = 0; k < SIZE * SIZE; ++k)
      total += img->n[layer][k];
  double scale = SIZE * SIZE * 1.0 / total;
  double scale2 = 255 * scale;
  scale = 1/log(1 + 1/scale);
  printf("%lu %e\n", total, scale);
  for (int layer = 0; layer < LAYERS; ++layer)
  {
    char filename[100];
    snprintf(filename, 100, "layer-%02d.pgm", layer);
    double l, a, b;
    { // convert from srgb to lab
      double r, g, bb;
      double x, y, z;
      printf("  %02d\r", layer);
      fflush(stdout);
      r  = spectrum[layer][0]/255.0;
      g  = spectrum[layer][1]/255.0;
      bb = spectrum[layer][2]/255.0;
//      printf("r g b: %f %f %f\n", r, g, bb);
      srgb2xyz(r, g, bb, &x, &y, &z);
//      printf("x y z: %f %f %f\n", x, y, z);
      xyz2lab(x, y, z, &l, &a, &b);
//      printf("l a b: %f %f %f\n", l, a, b);
//      lab2xyz(l, a, b, &x, &y, &z);
//      printf("x y z: %f %f %f\n", x, y, z);
//      xyz2srgb(x, y, z, &r, &g, &bb);
//      printf("r g b: %f %f %f\n", r, g, bb);
      l /= LAYERS;
      a /= LAYERS;
      b /= LAYERS;
    }
    #pragma omp parallel for
    for (Z j = 0; j < SIZE; ++j)
    for (Z i = 0; i < SIZE; ++i)
    {
      Z k = j * SIZE + i;
      double x = scale * log(1 + img->n[layer][k]);
#if 1
      double dither = ((((layer * 67 + j) * 236 + i) * 119) & 255) / 256.0;
```

```
        double y = 255 * x + dither;
        y = y > 255 ? 255 : y;
        ppm[k] = y;
#endif
        raw[3 * k + 0] += x * l;
        raw[3 * k + 1] += x * a;
        raw[3 * k + 2] += x * b;
      }
#if 1
      FILE *f = fopen(filename, "wb");
      fprintf(f, "P5\n%d %d\n255\n", SIZE, SIZE);
      fwrite(ppm, SIZE * SIZE, 1, f);
      fclose(f);
#endif
    }
    { // clamp
      #pragma omp parallel for
      for (Z k = 0; k < SIZE * SIZE; ++k)
      {
        double l = raw[3 * k + 0];
        if (l > 100)
        {
          double s = 100 / l;
          double a = raw[3 * k + 1];
          double b = raw[3 * k + 2];
          l *= s;
          a *= s;
          b *= s;
          raw[3 * k + 0] = l;
          raw[3 * k + 1] = a;
          raw[3 * k + 2] = b;
        }
      }
    }

#if 1
    {
      // auto white balance (make average colour white)
      double l = 0, a = 0, b = 0;
      #pragma omp parallel for reduction(+:l) reduction(+:a) reduction(+:b)
      for (Z k = 0; k < SIZE * SIZE; ++k)
      {
        l += raw[3 * k + 0];
        a += raw[3 * k + 1];
        b += raw[3 * k + 2];
      }
      l /= SIZE * SIZE;
      a /= SIZE * SIZE;
      b /= SIZE * SIZE;
      const double scale = 1.0 / 12 * 100 / l;
      const double shift = 1.0 / 100;
      #pragma omp parallel for
      for (Z k = 0; k < SIZE * SIZE; ++k)
      {
        double t = shift * raw[3 * k + 0];
        raw[3 * k + 1] -= a * t;
        raw[3 * k + 2] -= b * t;
```

```
        raw[3 * k + 0] *= scale;
        raw[3 * k + 1] *= scale;
        raw[3 * k + 2] *= scale;
      }
295   }
#endif

#if 1
  { // auto-levels
300   double quantile_lo = 0.001;
      double quantile_hi = 0.999;
      #pragma omp parallel for
      for (Z k = 0; k < SIZE * SIZE; ++k)
      {
305     double l = raw[3 * k + 0];
        double a = raw[3 * k + 1];
        double b = raw[3 * k + 2];
        double x, y, z;
        double r, g, bb;
310     lab2xyz(l, a, b, &x, &y, &z);
        xyz2srgb(x, y, z, &r, &g, &bb);
        raw[3 * k + 0] = r;
        raw[3 * k + 1] = g;
        raw[3 * k + 2] = bb;
315   }
      for (int c = 0; c < 3; ++c)
      {
        #pragma omp parallel for
        for (Z k = 0; k < SIZE * SIZE; ++k)
320       histogram[k] = raw[3 * k + c];
        qsort(histogram, SIZE * SIZE, sizeof(float), cmp_float);
        double lo = histogram[(Z) (SIZE * SIZE * quantile_lo)];
        double hi = histogram[(Z) (SIZE * SIZE * quantile_hi)];
        double range = 1.0 / (hi - lo);
325     if (range > 0) // nan check
        {
          #pragma omp parallel for
          for (Z k = 0; k < SIZE * SIZE; ++k)
          {
330         double v = raw[3 * k + c];
            v -= lo;
            v *= range;
            v = v < 0 ? 0 : v;
            v = v > 1 ? 1 : v;
335         raw[3 * k + c] = v;
          }
        }
      }
      #pragma omp parallel for
340   for (Z k = 0; k < SIZE * SIZE; ++k)
      {
        double r  = raw[3 * k + 0];
        double g  = raw[3 * k + 1];
        double bb = raw[3 * k + 2];
345     double x, y, z;
        double l, a, b;
        srgb2xyz(r, g, bb, &x, &y, &z);
```

```
        xyz2lab(x, y, z, &l, &a, &b);
        raw[3 * k + 0] = l;
        raw[3 * k + 1] = a;
        raw[3 * k + 2] = b;
      }
    }
#endif

#if 1
    { // increase saturation
      // https://math.stackexchange.com/questions/586424/adjust-saturation-in-cie-lab-space
      const double saturation = 2;
      const double limit = 0.95;
      #pragma omp parallel for
      for (Z k = 0; k < SIZE * SIZE; ++k)
      {
        double l = raw[3 * k + 0];
        double a = raw[3 * k + 1];
        double b = raw[3 * k + 2];
        double c = hypot(a, b);
        double s = c / hypot(l, c);
        s *= saturation;
        if (s > limit) s = limit;
        double t = s * l / sqrt((a * a + b * b) * (1 - s * s));
        if (t > 0) // nan check
        {
          a *= t;
          b *= t;
        }
        raw[3 * k + 1] = a;
        raw[3 * k + 2] = b;
      }
    }
#endif

#if 1
    {
      // auto white balance again (make average colour white)
      double l = 0, a = 0, b = 0;
      #pragma omp parallel for reduction(+:l) reduction(+:a) reduction(+:b)
      for (Z k = 0; k < SIZE * SIZE; ++k)
      {
        l += raw[3 * k + 0];
        a += raw[3 * k + 1];
        b += raw[3 * k + 2];
      }
      l /= SIZE * SIZE;
      a /= SIZE * SIZE;
      b /= SIZE * SIZE;
      const double scale = 1.0 / 12 * 100 / l * 5 / 2;
      const double shift = 1.0 / 100;
      #pragma omp parallel for
      for (Z k = 0; k < SIZE * SIZE; ++k)
      {
        double t = shift * raw[3 * k + 0];
```

```
        raw[3 * k + 1] -= a * t;
405     raw[3 * k + 2] -= b * t;
        raw[3 * k + 0] *= scale;
        raw[3 * k + 1] *= scale;
        raw[3 * k + 2] *= scale;
      }
410   }
#endif

  { // lab to 8bit srgb
    #pragma omp parallel for
415   for (Z j = 0; j < SIZE; ++j)
      for (Z i = 0; i < SIZE; ++i)
      {
        Z k = j * SIZE + i;
#if 1
420     double l = raw[3 * k + 0];
        double a = raw[3 * k + 1];
        double b = raw[3 * k + 2];
        double x, y, z, r, g;
        lab2xyz(l, a, b, &x, &y, &z);
425     xyz2srgb(x, y, z, &r, &g, &b);
#else
        double r = raw[3 * k + 0];
        double g = raw[3 * k + 1];
        double b = raw[3 * k + 2];
430 #endif
        double dither;
        dither = ((((0 * 67 + j) * 236 + i) * 119) & 255) / 256.0;
        ppm[3 * k + 0] = fmin(fmax(r * 255 + dither, 0), 255);
        dither = ((((1 * 67 + j) * 236 + i) * 119) & 255) / 256.0;
435     ppm[3 * k + 1] = fmin(fmax(g * 255 + dither, 0), 255);
        dither = ((((2 * 67 + j) * 236 + i) * 119) & 255) / 256.0;
        ppm[3 * k + 2] = fmin(fmax(b * 255 + dither, 0), 255);
      }
    }
440 FILE *f = fopen("colourized.ppm", "wb");
    fprintf(f, "P6\n%d %d\n255\n", SIZE, SIZE);
    fwrite(ppm, 3 * SIZE * SIZE, 1, f);
    fclose(f);
    free(ppm);
445 image_unmap(img);
}
```

10 buddhabrot/bsrenderlayers.c

```
// gcc -std=c99 -Wall -pedantic -Wextra -O3 -fopenmp -o bsrenderlayers ↙
    ↪ bsrenderlayers.c `PKG_CONFIG_PATH=${HOME}/opt/lib/pkgconfig pkg-config --↙
    ↪ cflags --libs mandelbrot-numerics` -lm
// dd if=/dev/zero of=bs.map bs=$((1024 * 1024)) count=512
// ./bsrenderlayers

5 #include <complex.h>
#include <math.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
```

```
#include <stdlib.h>
#include <string.h>
#include <signal.h>
#include <time.h>

typedef unsigned char B;
typedef float F;
typedef uint64_t N;
typedef int64_t Z;
typedef double R;
typedef double _Complex C;


#define unlikely(x) __builtin_expect(x, 0)


Z ilog2(Z n)
{
  Z l = 0;
  while (n > 0)
  {
    n >>= 1;
    l += 1;
  }
  return l;
}

#define LOG2SIZE 12
#define SIZE (1 << LOG2SIZE)
#define LAYERS 30

B mask[SIZE][SIZE];

struct image
{
  N n[LAYERS][SIZE * SIZE];
};


FILE *image_file = 0;


static struct image *image_map()
{
  bool mask_ok = false;
  FILE *mask_file = fopen("bs.pgm", "rb");
  int mask_width = 0;
  int mask_height = 0;
  if (2 == fscanf(mask_file, "P5\n%d %d\n255", &mask_width, &mask_height))
  {
    if ('\n' == fgetc(mask_file))
    {
      if (mask_width == SIZE && mask_height == SIZE)
      {
        if (1 == fread(&mask[0][0], SIZE * SIZE, 1, mask_file))
        {
          mask_ok = true;
```

```
        }
      }
    }
  }
  fclose(mask_file);
  if (! mask_ok)
  {
    memset(&mask[0][0], 255, SIZE * SIZE);
  }
  image_file = fopen("bs.map", "r+b");
  if (! image_file)
  {
    exit(1);
  }
  struct image *img = malloc(sizeof(struct image));
  if (! img)
  {
    exit(1);
  }
  fread(img, sizeof(struct image), 1, image_file);
  return img;
}

static void image_sync(struct image *img)
{
  rewind(image_file);
  fwrite(img, sizeof(struct image), 1, image_file);
}

static void image_unmap(struct image *img)
{
  image_sync(img);
  fclose(image_file);
  image_file = 0;
  free(img);
}

static inline void image_plot_with_bounds_checks(struct image *img, Z layer, R zx, R zy)
{
  // flipped along main diagonal
  Z y = floor(SIZE * (zx + 2.0) / 4.0);
  Z x = floor(SIZE * (zy + 2.0) / 4.0);
  if (0 <= x && x < SIZE && 0 <= y && y < SIZE)
  {
    Z k = (y << LOG2SIZE) + x;
    N *p = img->n[layer] + k;
    #pragma omp atomic
    *p += 1;
  }
}

static inline void image_plot_without_bounds_checks(struct image *img, Z layer,
```

```
    ↪ R zx, R zy)
{
  // flipped along main diagonal
  Z y = SIZE * (zx + 2.0) / 4.0;
  Z x = SIZE * (zy + 2.0) / 4.0;
  Z k = (y << LOG2SIZE) + x;
  N *p = img->n[layer] + k;
  #pragma omp atomic
  *p += 1;
}


static Z render(struct image *img, Z maxiters, Z grid, C mul, C add)
{
  Z total = 0;
  #pragma omp parallel for schedule(dynamic, 1) reduction(+:total)
  for (Z j = 0; j < grid; ++j)
  {
    Z stotal = 0;
    for (Z i = 0; i < grid; ++i)
    {
      C c = mul * (i + I * j) + add;
      R cx = creal(c);
      R cy = cimag(c);
      Z mi = floor(SIZE * (cy + 2.0) / 4.0);
      Z mj = floor(SIZE * (cx + 2.0) / 4.0);
      if (0 <= mi && mi < SIZE && 0 <= mj && mj < SIZE)
        if (mask[mi][mj] != 255)
          continue;
      R zx = 0;
      R zy = 0;
      R zx2 = zx * zx;
      R zy2 = zy * zy;
      R zxy = fabs(zx * zy);
      Z count;
      for (count = 1; count < maxiters; ++count)
      {
        zx = zx2 - zy2 + cx;
        zy = zxy + zxy + cy;
        zx2 = zx * zx;
        zy2 = zy * zy;
        zxy = fabs(zx * zy);
        if (unlikely(zx2 + zy2 > 4))
          break;
      }
      if (count < maxiters)
      {
        zx = 0;
        zy = 0;
        cx = creal(c);
        cy = cimag(c);
        Z layer = ilog2(count - 1);
        if (! (0 <= layer && layer < LAYERS))
          continue;
        zx2 = zx * zx;
        zy2 = zy * zy;
        zxy = fabs(zx * zy);
```

```
            Z n;
            for (n = 1; n < count; ++n)
            {
              zx = zx2 - zy2 + cx;
              zy = zxy + zxy + cy;
              zx2 = zx * zx;
              zy2 = zy * zy;
              zxy = fabs(zx * zy);
              image_plot_without_bounds_checks(img, layer, zx, zy);
            }
            while (zx2 + zy2 < 16.0 && n++ < count + 16)
            {
              zx = zx2 - zy2 + cx;
              zy = zxy + zxy + cy;
              zx2 = zx * zx;
              zy2 = zy * zy;
              zxy = fabs(zx * zy);
              image_plot_with_bounds_checks(img, layer, zx, zy);
            }
            stotal += count;
          }
      }
      total = total + stotal;
    }
    return total;
}

static volatile bool running = true;


static void handler(int sig)
{
  (void) sig;
  running = false;
}

extern int main(int argc, char **argv) {
  (void) argc;
  (void) argv;
  Z seed = time(0);
  fprintf(stdout, "starting with seed %016lx\n", seed);
  fflush(stdout);
  srand(seed);
  Z depth = LAYERS;
  Z maxiters = 1LL << depth;
  Z pixels = 0;
  Z total = 0;
  struct image *img = image_map();
  time_t last = time(0);
  signal(SIGINT, handler);
  signal(SIGTERM, handler);
  while (running)
  {
    // randomize grid
    Z n = exp2(10 + (rand() / (R) RAND_MAX));
```

```
    n += (n & 1); // ensure even
    R l = 2.0 / (n/2 - 1) + (2.0/(n/2-3) - 2.0/(n/2-1)) * (rand() / (R) RAND_MAX↙
        ↳ );
    l *= sqrt(2);
    R x = (rand() / (R) RAND_MAX);
    R y = (rand() / (R) RAND_MAX);
    R t = 2 * 3.141592653589793 * (rand() / (R) RAND_MAX);
    C mul = l * cexp(I * t);
    C add = - mul * ((n/2 + x) + I * ((n/2) + y));
    // calculate
    total += render(img, maxiters, n, mul, add);
    pixels += n * n;
    // output
    fprintf(stdout, "%18ld / %18ld = %.18f\n", total, pixels, total / (double) ↙
        ↳ pixels);
    fflush(stdout);
    time_t now = time(0);
    if (now - last >= 60) // 1hr
    {
      fprintf(stdout, "syncing...\n");
      fflush(stdout);
      image_sync(img);
      last = time(0);
      fprintf(stdout, "...synced\n");
      fflush(stdout);
    }
  }
  fprintf(stdout, "exiting\n");
  fflush(stdout);
  image_unmap(img);
  return 0;
}


// scalar 1476008714
//        1477998632
// END
```

11 buddhabrot/cusp.c

```
// gcc -std=c99 -Wall -pedantic -Wextra -O3 -lm -o cusp cusp.c
// ./cusp | tee cusp.dat

#include <complex.h>
#include <math.h>
#include <stdio.h>

typedef unsigned int N;
typedef double R;
typedef double _Complex C;

static inline R cabs2(C z) { return creal(z) * creal(z) + cimag(z) * cimag(z); }

static void render(N depth) {
  const R pi = 3.141592653589793;
  C c = -0.75 + I * pow(0.5, depth);
  R a = 0;
```

```
  C z = 0;
  C dc = 0;
  printf("%d %.18e %.18e ", depth, creal(c), cimag(c));
  fflush(stdout);
  while (cabs2(z) < 65536)
  {
    a = a + 1;
    dc = 2 * z * dc + 1;
    z = z * z + c;
  }
  R de = 2 * cabs(z) * log(cabs(z)) / cabs(dc);
  printf("%.18e %.18e ", a, de);
  a = pi * de * de * a;
  printf("%.18e\n", a);
  fflush(stdout);
}

extern int main(int argc, char **argv) {
  (void) argc;
  (void) argv;
  for (N depth = 0; 1; ++depth)
    render(depth);
  return 0;
}

// END
```

12 buddhabrot/expectmu.c

```
// gcc -std=c99 -Wall -pedantic -Wextra -O3 -fopenmp -o expectmu expectmu.c -lm

#include <assert.h>
#include <complex.h>
#include <math.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>

typedef int64_t Z;
typedef double R;
typedef double _Complex C;

#define unlikely(x) __builtin_expect(x, 0)

struct partial {
  C z;
  Z p;
};

struct compute {
  Z tag, bias;
  struct partial *partials;
  Z npartials, np, n;
  R er2, mz2;
  C c, z;
```

```
};

static struct compute *compute_new(Z npartials) {
  struct compute *px = malloc(sizeof(*px));
  if (! px) {
    return 0;
  }
  if (npartials > 0) {
    px->partials = malloc(npartials * sizeof(*(px->partials)));
    px->npartials = npartials;
    if (! px->partials) {
      free(px);
      return 0;
    }
  } else {
    px->partials = 0;
    px->npartials = 0;
  }
  px->tag = 0;
  return px;
}

static void compute_delete(struct compute *px) {
  if (px) {
    if (px->partials) {
      free(px->partials);
    }
    free(px);
  }
}

static void compute_init(struct compute *px, Z bias, C c) {
  if (! px) { return; }
  px->tag = 0;
  px->bias = bias;
  px->mz2 = 1.0 / 0.0;
  px->c = c;
  px->z = 0;
  px->n = 0;
  px->np = 0;
}

static bool cisfinite(C z)
{
  return isfinite(creal(z)) && isfinite(cimag(z));
}

static Z attractor_step(C *z, C z_guess, C c, Z period) {
  // one newton step for solving z = f^p(z, c)
  R epsilon = nextafter(1, 2) - 1;
  C zz = z_guess;
  C dzz = 1;
  for (Z i = 0; i < period; ++i) {
    dzz = 2 * zz * dzz;
    zz = zz * zz + c;
  }
  if (cabs(zz - z_guess) <= epsilon) {
```

```
85      *z = z_guess;
        return 0; // converged
      }
      C z_new = z_guess - (zz - z_guess) / (dzz - 1);
      C d = z_new - zz;
90    if (cabs(d) <= epsilon) {
        *z = z_new;
        return 0; // converged
      }
      if (cisfinite(d)) {
95      *z = z_new;
        return 1; // stepped
      } else {
        *z = z_guess;
        return -1; // failed
100   }
    }

    static Z attractor(C *z_out, C z_guess, C c, Z period, Z maxsteps) {
      // newton's method
105   Z result = -1;
      C z = z_guess;
      for (Z i = 0; i < maxsteps; ++i) {
        if (1 != (result = attractor_step(&z, z, c, period))) {
          break;
110     }
      }
      *z_out = z;
      return result;
    }
115
    static bool is_interior(C c, C z, Z p, Z steps)
    {
      // find attractor
      C z00 = 0;
120   if (-1 != attractor(&z00, z, c, p, steps)) {
        // compute derivative
        C z0 = z00;
        C dz0 = 1;
        for (Z j = 0; j < p; ++j)
125     {
          dz0 = 2.0 * z0 * dz0;
          z0 = z0 * z0 + c;
        }
        // numerator of interior distance forumula is positive?
130     return cabs(dz0) <= 1;
      }
      return false;
    }

135 static bool compute_step(struct compute *px, Z steps)
    {
      static const double er2 = 4;
      if (! px) {
        return false;
140   }
      if (px->tag != 0) {
```

```
      return true;
    }
    px->np = 0;
    // load state
    C c = px->c;
    C z = px->z;
    R mz2 = px->mz2;
    Z n = px->n;
    R cx = creal(c);
    R cy = cimag(c);
    R zx = creal(z);
    R zy = cimag(z);
    R zx2 = zx * zx;
    R zy2 = zy * zy;
    R zxy = zx * zy;
    // if last pixel was interior, perform interior checks as we go
    if (px->bias < 0)
    {
      for (Z i = 1; i <= steps; ++i)
      {
        // step
        zx = zx2 - zy2 + cx;
        zy = zxy + zxy + cy;
        zx2 = zx * zx;
        zy2 = zy * zy;
        zxy = zx * zy;
        R z2 = zx2 + zy2;
        if (unlikely(z2 < mz2))
        {
          // check interior
          mz2 = z2;
          if (is_interior(c, zx + I * zy, n + i, 64))
          {
            px->tag = -1;
            return true;
          }
        }
        if (unlikely(z2 >= er2))
        {
          // escaped
          px->tag = 1;
          px->n = n + i;
          px->z = zx + I * zy;
          return true;
        }
      }
      // save state, no conclusions yet
      px->tag = 0;
      px->z = zx + I * zy;
      px->mz2 = mz2;
      px->n = n + steps;
      return false;
    }
    // if last pixel was not interior, postpone interior checks to the end
    else
    {
      for (Z i = 1; i <= steps; ++i)
```

```
    {
200   // step
      zx = zx2 - zy2 + cx;
      zy = zxy + zxy + cy;
      zx2 = zx * zx;
      zy2 = zy * zy;
205   zxy = zx * zy;
      R z2 = zx2 + zy2;
      if (unlikely(z2 < mz2))
      {
        // save for later
210     mz2 = z2;
        if (px->partials && px->np < px->npartials)
        {
          px->partials[px->np].z = zx + I * zy;
          px->partials[px->np].p = n + i;
215       px->np = px->np + 1;
        }
      }
      if (unlikely(z2 >= er2))
      {
220     // escaped
        px->tag = 1;
        px->n = n + i;
        px->z = zx + I * zy;
        return true;
225   }
    }
    // perform postponed interior checks
    for (Z i = 0; i < px->np; ++i)
    {
230   // check interior
      z = px->partials[i].z;
      Z p = px->partials[i].p;
      if (is_interior(c, z, p, 64))
      {
235     px->tag = -1;
        px->z = z;
        px->n = p;
        return true;
      }
240 }
    // save state, no conclusions yet
    px->tag = 0;
    px->z = zx + I * zy;
    px->mz2 = mz2;
245 px->n = n + steps;
    return false;
  }
  return false;
}
250
static void render(Z grid, C mul, C add, Z *o_s0, R *o_s1)
{
  R smu = 0;
  Z scount = 0;
255 #pragma omp parallel for schedule(dynamic, 1) reduction(+:smu) reduction(+:↙
```

```
      ↪ scount)
  for (Z j = 0; j < grid; ++j)
  {
    R mu = 0;
    Z count = 0;
    // allocate space for at most 1<<24 partials (for interiority checks)
    struct compute *px = compute_new(1 << 24);
    for (Z i = 0; i < grid; ++i)
    {
      C c = mul * (i + I * j) + add;
      // only consider points inside D(0,2), reject real axis part and pre-↙
          ↪ periodic 0+1i
      if (cabs(c) <= 2 && !(cimag(c) == 0 && -2 <= creal(c) && creal(c) <= 0.25)↙
          ↪  && !(c == I))
      {
        compute_init(px, px->tag, c);
        // performs interiority checks after every doubling of iterations
        for (Z log2iters = 10; 1; log2iters += 1)
        {
          assert(log2iters < 63); // FIXME
          compute_step(px, ((Z)1) << log2iters);
          if (px->tag != 0)
            break;
        }
        if (px->tag > 0)
        {
          R u = px->n;
          if (u > 0)
          {
            // accumulate output
            mu += u;
            count += 1;
          }
        }
      }
    }
    compute_delete(px);
    smu = smu + mu;
    scount = scount + count;
  }
  *o_s0 += scount;
  *o_s1 += smu;
}

extern int main()
{
  // set random seed
  // TOOD use a reproducible generator from libgsl or similar
  srand(0x1cedcafe);
  Z s0 = 0;
  R s1 = 0;
  R s2 = 0;
  while (1)
  {
    Z ls0 = 0;
    R ls1 = 0;
    // accumulate at least 1<<30 points from complement(M) `intersect` D(0,2)
```

```
310       while (ls0 < 1 << 30)
          {
            // randomize grid
            Z n = exp2(13 + (rand() / (R) RAND_MAX));
            n += (n & 1); // ensure even
315         R l = 2.0 / (n/2 - 1) + (2.0/(n/2-3) - 2.0/(n/2-1)) * (rand() / (R) ↙
                ↳ RAND_MAX);
            l *= 3.6;
            R x = (rand() / (R) RAND_MAX);
            R y = (rand() / (R) RAND_MAX);
            R t = 2 * 3.141592653589793 * (rand() / (R) RAND_MAX);
320         C mul = l * cexp(I * t);
            C add = - mul * ((n/2 + x) + I * ((n/2) + y));
            // calculate
            render(n, mul, add, &ls0, &ls1);
            fprintf(stderr, "%4d\r", (int) (ls0 * 100.0 / (1 << 30)));
325       }
          // output statistics
          R s = ls1 / ls0;
          s0 += 1;
          s1 += s;
330       s2 += s * s;
          R mean = s1 / s0;
          R stddev = sqrt((s0 * s2 - s1 * s1) / (s0 * (s0 - 1)));
          fprintf(stdout, "%.20f %.20f %.20f\n", s, mean, stddev);
          fflush(stdout);
335     }
        return 0;
      }

      // END
```

13 buddhabrot/histogram.c

```
      #include <math.h>
      #include <stdio.h>
      #include <stdlib.h>
      #include <string.h>
 5
      int main(int argc, char **argv)
      {
        (void) argc;
        (void) argv;
10      int w = 0;
        int h = 0;
        int n = scanf("P5\n%d %d\n255", &w, &h);
        int c = fgetc(stdin);
        if (n == 2 && c == '\n' && w > 0 && h > 0)
15      {
          unsigned char *pgm = malloc(w * h);
          fread(pgm, w * h, 1, stdin);
          int *hist = calloc(1, 256 * sizeof(*hist));
          for (int k = 0; k < w * h; ++k)
20          hist[pgm[k]]++;
          unsigned char *out = malloc(128 *  256);
          memset(out, 255, 128 * 256);
          double scale = 128 / log(1 + w * h);
```

```
    for (int i = 0; i < 256; ++i)
      {
        int y = log(1 + hist[i]) * scale;
        for (int j = 0; j < y && j < 128; ++j)
          out[(127 - j) * 256 + i] = 0;
      }
    printf("P5\n%d %d\n255\n", 256, 128);
    fwrite(out, 256 * 128, 1, stdout);
  }
  return 0;
}
```

14 buddhabrot/Makefile

```
all: bb bbrender bbrenderlayers bsrenderlayers bbcolourizelayers ↙
    ↪ bscolourizelayers bound bound-2 cusp tip histogram

bb: bb.c
	gcc -std=c99 -Wall -pedantic -Wextra -O3 -fopenmp -lm -o bb bb.c

bbrender: bbrender.c
	gcc -std=c99 -Wall -pedantic -Wextra -O3 -fopenmp -o bbrender bbrender.c↙
	    ↪  `PKG_CONFIG_PATH=${HOME}/opt/lib/pkgconfig pkg-config --cflags --↙
	    ↪ libs mandelbrot-graphics` -lm

bbrenderlayers: bbrenderlayers.c
	gcc -std=c99 -Wall -pedantic -Wextra -O3 -fopenmp -o bbrenderlayers ↙
	    ↪ bbrenderlayers.c `PKG_CONFIG_PATH=${HOME}/opt/lib/pkgconfig pkg-↙
	    ↪ config --cflags --libs mandelbrot-numerics` -lm -g

bsrenderlayers: bsrenderlayers.c
	gcc -std=c99 -Wall -pedantic -Wextra -O3 -fopenmp -march=native -o ↙
	    ↪ bsrenderlayers bsrenderlayers.c -lm -g

bbcolourizelayers: bbcolourizelayers.c
	gcc -std=c99 -Wall -pedantic -Wextra -O3 -fopenmp -o bbcolourizelayers ↙
	    ↪ bbcolourizelayers.c -lm

bscolourizelayers: bscolourizelayers.c
	gcc -std=c99 -Wall -pedantic -Wextra -O3 -fopenmp -o bscolourizelayers ↙
	    ↪ bscolourizelayers.c -lm

bound: bound.c
	gcc -std=c99 -Wall -pedantic -Wextra -O3 -fopenmp -o bound bound.c `↙
	    ↪ PKG_CONFIG_PATH=${HOME}/opt/lib/pkgconfig pkg-config --cflags --↙
	    ↪ libs mandelbrot-graphics`

bound-2: bound-2.c
	gcc -std=c99 -Wall -pedantic -Wextra -O3 -fopenmp -o bound-2 bound-2.c `↙
	    ↪ PKG_CONFIG_PATH=${HOME}/opt/lib/pkgconfig pkg-config --cflags --↙
	    ↪ libs mandelbrot-graphics` -lm

cusp: cusp.c
	gcc -std=c99 -Wall -pedantic -Wextra -O3 -lm -o cusp cusp.c

tip: tip.c
	gcc -std=c99 -Wall -pedantic -Wextra -O3 -lm -o tip tip.c
```

```
histogram: histogram.c
	gcc -std=c99 -Wall -pedantic -Wextra -O3 -lm -o histogram histogram.c

expectmu: expectmu.c
	gcc -std=c99 -Wall -pedantic -Wextra -O3 -fopenmp -o expectmu expectmu.c
	↳ -lm -g
```

15 buddhabrot/spectrum.ppm

16 buddhabrot/tip.c

```
// gcc -std=c99 -Wall -pedantic -Wextra -O3 -lm -o tip tip.c
// ./tip | tee tip.dat

#include <complex.h>
#include <math.h>
#include <stdio.h>

typedef unsigned int N;
typedef double R;
typedef double _Complex C;

static inline R cabs2(C z) { return creal(z) * creal(z) + cimag(z) * cimag(z); }

static void render(N depth) {
  const R pi = 3.141592653589793;
  C c = -2 - pow(0.5, depth);
  R a = 0;
  C z = 0;
  C dc = 0;
  printf("%d %.18e %.18e ", depth, creal(c), cimag(c));
  fflush(stdout);
  while (cabs2(z) < 65536)
  {
    a = a + 1;
    dc = 2 * z * dc + 1;
    z = z * z + c;
  }
  R de = 2 * cabs(z) * log(cabs(z)) / cabs(dc);
  printf("%.18e %.18e ", a, de);
  a = pi * de * de * a;
  printf("%.18e\n", a);
  fflush(stdout);
}


extern int main(int argc, char **argv) {
  (void) argc;
  (void) argv;
  for (N depth = 0; 1; ++depth)
    render(depth);
  return 0;
}
```

```
// END
```

17 burning-ship-series-approximation/BossaNova2.cxx

```
/*
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE FlexibleContexts #-}

import Control.Concurrent.Spawn (parMapIO_)
import Control.Monad (join)
import Control.Monad.ST (runST)
import Control.Parallel.Strategies (parMap, rseq)
import Data.List (minimumBy, nub, transpose)
import Data.Map.Strict (Map)
import qualified Data.Map.Strict as M
import Data.Monoid (Sum(..))
import Data.Ord (comparing)
import Data.Vector.Storable (unsafeFreeze, unsafeWith)
import qualified Data.Vector.Storable.Mutable as S
import qualified Data.Vector.Unboxed.Mutable  as U
import Data.Word (Word8)
import Numeric.Rounded (Rounded, RoundingMode(TowardNearest), Precision, ↙
    ↳ reifyPrecision)
import System.Environment (getArgs)
import System.IO (hPutBuf, hPutStr, stdout, stderr)
import Debug.Trace (trace, traceShow)

diffabs :: (Ord a, Num a) => a -> a -> a
diffabs c d =
  if (c >= 0)
    then
      if (c + d >= 0)
        then d
        else -(2 * c + d)
    else
      if (c + d > 0)
        then 2 * c + d
        else -d
*/

#include <cassert>
#include <cstdint>
#include <cstdio>
#include <iostream>
#include <map>
#include <optional>
#include <set>
#include <vector>

#include <mpreal.h>

template <typename T>
T diffabs(const T &c, const T &d)
{
  if (c >= 0) if (c + d >= 0) return d; else return -(2 * c + d);
  else if (c + d > 0) return 2 * c + d; else return -d;
```

```
}

/*
er :: Double
er = 65536

er2 :: Double
er2 = er^2

type REAL p = Rounded TowardNearest p
*/

typedef uint8_t N_8;
typedef uint16_t N_16;
typedef int64_t Z_64;
typedef double R_lo;
typedef mpfr::mpreal R_hi;

/*
const R_lo er = 65536;
const R_lo er2 = er * er;
*/

typedef std::pair<N_16,N_16> Coord;

/*
neighbours (i, j) = [ (i - 1, j), (i + 1, j), (i, j - 1), (i, j + 1) ]
*/

std::vector<Coord> neighbours(const Coord &ij)
{
  std::vector<Coord> r;
  auto i = ij.first;
  auto j = ij.second;
  r.push_back(Coord(i-1,j));
  r.push_back(Coord(i+1,j));
  r.push_back(Coord(i,j+1));
  r.push_back(Coord(i,j-1));
  return r;
}

/*
data Reference p = Reference{ aa, bb, xx, yy :: !(REAL p), n :: !Int }
*/

/**
struct Reference
{
  R_hi a, b, x, y;
  Z_64 n;
  Reference(R_hi a, R_hi b, R_hi x, R_hi y, Z_64 n) : a(a), b(b), x(x), y(y), n(↙
      ↳ n) { };
};

Reference burningShipRef(const Reference &r)
{
  return Reference(r.a, r.b, sqr(r.x) - sqr(r.y) + r.a, abs(2 * r.x * r.y) + r.b↙
```

```
      ↪ , r.n + 1);
}
*/

struct Orbit
{
  std::vector<R_lo> x, y, g;
  Orbit(R_hi a, R_hi b, Z_64 maxiters, R_lo glitch_threshold, R_lo er2)
  {
    R_hi zx = 0;
    R_hi zy = 0;
    x.reserve(maxiters);
    y.reserve(maxiters);
    g.reserve(maxiters);
    for (auto n = 0; n < maxiters; ++n)
    {
      R_lo rx(zx);
      R_lo ry(zy);
      R_lo r2 = (rx * rx + ry * ry) * glitch_threshold;
      x.push_back(rx);
      y.push_back(ry);
      g.push_back(r2);
      if (r2 > er2)
        break;
      R_hi xn = sqr(zx) - sqr(zy) + a;
      R_hi yn = abs(2 * zx * zy) + b;
      zx = xn;
      zy = yn;
    }
    x.shrink_to_fit();
    y.shrink_to_fit();
    g.shrink_to_fit();
  }
};

/*
data Delta = Delta{ a, b, x, y :: !Double }

toDelta :: Precision p => Reference p -> Delta
toDelta r = Delta
  { a = realToFrac (aa r)
  , b = realToFrac (bb r)
  , x = realToFrac (xx r)
  , y = realToFrac (yy r)
  }
*/

struct Delta
{
  R_lo a, b, x, y;
  Delta(R_lo a, R_lo b, R_lo x, R_lo y) : a(a), b(b), x(x), y(y) {};
//  Delta(const Reference &r) : a(r.a), b(r.b), x(r.x), y(r.y) {};
  Delta() : a(0), b(0), x(0), y(0) {};
};

/*
burningShipDel :: Delta -> Delta -> Delta
```

```
burningShipDel r d = d
  { x = 2 * x r * x d - 2 * y r * y d + x d ^ 2 - y d ^ 2 + a d
  , y = 2 * diffabs (x r * y r) (x r * y d + x d * y r + x d * y d) + b d
  }
*/

Delta burningShipDel(const Delta &r, const Delta &d)
{
  return Delta
    ( d.a
    , d.b
    , (2 * r.x + d.x) * d.x - (2 * r.y + d.y) * d.y + d.a
    , 2 * diffabs(r.x * r.y, r.x * d.y + d.x * r.y + d.x * d.y) + d.b
    );
}

enum Quadrant
{ Q0
, Q1
, Q2
, Q3
};

/*
burningShipFolding :: Delta -> Delta -> Int
burningShipFolding r d = diffabsIx (x r * y r) (x r * y d + x d * y r + x d * y ↙
    ↳ d)
{-
  | x r + x d >= 0 && y r + y d >= 0 = 0
  | x r + x d <  0 && y r + y d >= 0 = 1
  | x r + x d <  0 && y r + y d <  0 = 2
  | x r + x d >= 0 && y r + y d <  0 = 3
  | otherwise = traceShow (x r, x d, x r + x d, y r, y d, y r + y d) 0
-}

diffabsIx :: (Ord a, Num a) => a -> a -> Int
diffabsIx c d =
  if (c >= 0)
    then
      if (c + d >= 0)
        then 0
        else 1
    else
      if (c + d > 0)
        then 2
        else 3
*/

Quadrant burningShipFolding(const Delta &r, const Delta &d)
{
  R_lo c = r.x * r.y;
  R_lo cd = c + r.x * d.y + d.x * r.y + d.x * d.y;
  if (c >= 0)
    if (cd >= 0)
      return Q0;
    else
      return Q1;
```

```
    else
      if (cd > 0)
        return Q2;
      else
        return Q3;
}

Quadrant burningShipFolding(const Delta &r) // specialized for d = 0 0
{
  if (r.x * r.y >= 0) return Q0; else return Q3;
}

typedef double Result;
const Result Unescaped          =  0;
const Result ReferenceEscaped   = -1;
const Result PerturbationGlitch = -2;

/*
result m d = Result $ fromIntegral m + 1 - log (log (sqrt (x d ^ 2 + y d ^ 2))) /
    ↳ / log 2
*/

Result result(Z_64 m, R_lo x, R_lo y)
{
  return m + 1 - log2(log(sqrt(x * x + y * y)));
}


/*
type BiSeries = Map (Int, Int) Double
*/

typedef std::map<Coord,R_lo> BiSeries;

/*
zero order = M.fromList [((i,j), 0) | i <- [0..order], j <- [0..order], i + j <=/
    ↳  order ]
*/

BiSeries ZeroSeries(N_16 order)
{
  BiSeries s;
  for (auto i = 0; i <= order; ++i)
    for (auto j = 0; j <= order - i; ++j)
      s[Coord(i,j)] = 0;
  return s;
}

/*
biSeries1 s a b = sum [ e * a^i * b^j | ((i, j), e) <- M.assocs s ]
*/

R_lo biSeries1(N_16 order, const BiSeries &s, R_lo a, R_lo b)
{
  R_lo an[order + 1];
  R_lo bn[order + 1];
  an[0] = 1;
```

```
275   bn[0] = 1;
      for (auto i = 1; i <= order; ++i)
      {
        an[i] = an[i-1] * a;
        bn[i] = bn[i-1] * b;
280   }
      R_lo total = 0;
      for (auto i = 0; i <= order; ++i)
      {
        R_lo subtotal = 0;
285     for (auto j = 0; j <= order - i; ++j)
          subtotal += s.at(Coord(i,j)) * bn[j];
        total += subtotal * an[i];
      }
      return total;
290 }

    /*
    biSeries (s, t) d = d{ x = biSeries1 s (ad) (bd), y = biSeries1 t (ad) (bd) }
      where ad = a d
295         bd = b d
    */

    Delta biSeries(N_16 order, const BiSeries &s, const BiSeries &t, const Delta &d)
    {
300   return Delta
        ( d.a
        , d.b
        , biSeries1(order, s, d.a, d.b)
        , biSeries1(order, t, d.a, d.b)
305     );
    }

    /*
    burningShipSeries order q r (s, t) = (s', t')
310   where
        s' = M.fromList [((i,j),
          (if (i,j) == (1,0) then 1 else 0) +
          2 * xr * s M.! (i,j) - 2 * yr * t M.! (i,j) +
          sum [ s M.! (k, l) * s M.! (i - k, j - l)
315             - t M.! (k, l) * t M.! (i - k, j - l)
                | k <- [0..i], l <- [0..j] ])
          | i <- [0..order], j <- [0..order], i + j <= order ]
        t' = M.fromList [((i,j),
          (if (i,j) == (0,1) then 1 else 0) +
320       case q of
            0 ->  2 * (xr * t M.! (i, j) + s M.! (i, j) * yr + sum [ s M.! (k, l) * ↙
                ↳ t M.! (i - k, j - l) | k <- [0..i], l <- [0..j] ])
            1 -> -2 * (xr * t M.! (i, j) + s M.! (i, j) * yr + sum [ s M.! (k, l) * ↙
                ↳ t M.! (i - k, j - l) | k <- [0..i], l <- [0..j] ] + if (i,j) == ↙
                ↳ (0,0) then 2 * xr * yr else 0)
            2 ->  2 * (xr * t M.! (i, j) + s M.! (i, j) * yr + sum [ s M.! (k, l) * ↙
                ↳ t M.! (i - k, j - l) | k <- [0..i], l <- [0..j] ] + if (i,j) == ↙
                ↳ (0,0) then 2 * xr * yr else 0)
            3 -> -2 * (xr * t M.! (i, j) + s M.! (i, j) * yr + sum [ s M.! (k, l) * ↙
                ↳ t M.! (i - k, j - l) | k <- [0..i], l <- [0..j] ])
325       )
```

```
      | i <- [0..order], j <- [0..order], i + j <= order ]
    xr = x r
    yr = y r
*/

std::pair<BiSeries, BiSeries> burningShipSeries(N_16 order, Quadrant q, const ↙
    ↳ Delta &r, const BiSeries &s, const BiSeries &t)
{
  BiSeries sn, tn;
  for (auto i = 0; i <= order; ++i)
  {
    for (auto j = 0; j <= order - i; ++j)
    {
      const Coord ij(i, j);
      // s
      R_lo total = 0;
      if (i == 1 && j == 0) total += 1;
      total += 2 * (r.x * s.at(ij) - r.y * t.at(ij));
      for (auto k = 0; k < i; ++k)
      {
        const auto ik = i - k;
        for (auto l = 0; l < j; ++l)
        {
          const auto jl = j - l;
          const Coord kl(k, l);
          const Coord ikjl(ik, jl);
          total += s.at(kl) * s.at(ikjl) - t.at(kl) * t.at(ikjl);
        }
      }
      sn[ij] = total;
      // t;
      total = 0;
      total += r.x * t.at(ij) + r.y * s.at(ij);
      for (auto k = 0; k < i; ++k)
      {
        const auto ik = i - k;
        for (auto l = 0; l < j; ++l)
        {
          const auto jl = j - l;
          const Coord kl(k, l);
          const Coord ikjl(ik, jl);
          total += s.at(kl) * t.at(ikjl);
        }
      }
      if (i == 0 && j == 0 && (q == Q1 || q == Q2)) total += 2 * r.x * r.y;
      total *= 2;
      if (q == Q1 || q == Q3) total = -total;
      if (i == 0 && j == 1) total += 1;
      tn[ij] = total;
    }
  }
  return std::pair<BiSeries,BiSeries>(sn, tn);
}

/*
data Region p = RegionSingleton{ ref :: !(Reference p), ix :: !(Int, Int) }
              | RegionSeries{ ref :: !(Reference p), s, t :: !(BiSeries), interior ↙
```

```
                ↳ , boundary :: !(Map (Int, Int) Delta) }
              | RegionPerturb{ ref :: !(Reference p), active :: !(Map (Int, Int) ↙
                ↳ Delta) }
              | RegionFinished{ done :: !(Map (Int, Int) Result) }
*/

enum RegionType
{
  RegionTypeFinished,
/**
  RegionTypeSingleton,
*/
  RegionTypePerturb,
  RegionTypeSeries
};

typedef std::map<Coord,Delta> Deltas;

typedef std::map<Coord,Result> RegionFinished;

/**
struct RegionSingleton
{
  const Orbit &orbit;
  Z_64 n;
  Coord ij;
  RegionSingleton(const Orbit &orbit, Z_64 n, Coord ij) : orbit(orbit), n(n), ij↙
      ↳ (ij) { };
};
*/

struct RegionPerturb
{
  const Orbit orbit;
  Z_64 n;
  Deltas active;
  RegionPerturb(const Orbit &orbit, Z_64 n, const Deltas &active) : orbit(orbit)↙
      ↳ , n(n), active(active) { };
};

struct RegionSeries
{
  const Orbit orbit;
  Z_64 n;
  BiSeries s, t;
  Deltas interior, boundary;
  RegionSeries
  ( const Orbit &orbit
  , const Z_64 &n
  , const BiSeries &s
  , const BiSeries &t
  , const Deltas &interior
  , const Deltas &boundary
  )
  : orbit(orbit), n(n), s(s), t(t), interior(interior), boundary(boundary)
  { };
};
```

```
435
struct Region
{
  RegionType tag;
  std::optional<RegionFinished> Finished;
440 /**
  std::optional<RegionSingleton> Singleton;
*/
  std::optional<RegionPerturb> Perturb;
  std::optional<RegionSeries> Series;
445   Region(const RegionFinished  &r) : tag(RegionTypeFinished),  Finished (r) { };
/**
  Region(const RegionSingleton &r) : tag(RegionTypeSingleton), Singleton(r) { };
*/
  Region(const RegionPerturb   &r) : tag(RegionTypePerturb),   Perturb  (r) { };
450   Region(const RegionSeries    &r) : tag(RegionTypeSeries) ,   Series   (r) { };
};

/*
size :: Region p -> (Char, Int)
455 size r@(RegionSingleton{}) = ('1', 1)
size r@(RegionSeries{}) = ('S', M.size (interior r) + M.size (boundary r))
size r@(RegionPerturb{}) = ('P', M.size (active r))
size r@(RegionFinished{}) = ('F', M.size (done r))
*/
460

Z_64 size(const RegionFinished &r) { return r.size(); }
/**
Z_64 size(const RegionSingleton &r) { (void) r; return 1; }
465 */
Z_64 size(const RegionPerturb &r) { return r.active.size(); }
Z_64 size(const RegionSeries &r) { return r.interior.size() + r.boundary.size();↙
    ↪  }

Z_64 size(const Region &r)
470 {
  switch (r.tag)
  {
    case RegionTypeFinished:  return size(r.Finished .value());
/**
475     case RegionTypeSingleton: return size(r.Singleton.value());
*/
    case RegionTypePerturb:   return size(r.Perturb  .value());
    case RegionTypeSeries:    return size(r.Series   .value());
  }
480   assert(!"size(Region) tag");
}

/*
interpolate r = (biSeries (s r, t r) <$> interior r) `M.union` boundary r
485 */

Deltas interpolate(N_16 order, const RegionSeries &r)
{
  Deltas rs;
490   for (auto d : r.boundary)
```

```
    rs[d.first] = d.second;
  for (auto d : r.interior)
    rs[d.first] = biSeries(order, r.s, r.t, d.second);
  return rs;
}




/*
initial :: Precision p => Int -> Int -> Int -> REAL p -> REAL p -> Double -> Region p
initial w h order a0 b0 r0 = RegionSeries
  { ref = Reference a0 b0 0 0 0
  , s = zero order
  , t = zero order
  , interior = int
  , boundary = bry
  }
  where
    (bry, int) = M.partitionWithKey (\(i, j) _ -> i == 0 || j == 0 || i == w - 1 || j == h - 1) deltas
    deltas = M.fromList
      [ ((i, j), Delta u v 0 0)
      | i <- [0 .. w - 1]
      , j <- [0 .. h - 1]
      , let v = 2 * r0 * ((fromIntegral j + 0.5) / fromIntegral h - 0.5)
      ]
*/

RegionSeries initial(N_16 w, N_16 h, Z_64 maxiters, N_16 order, R_lo glitch_threshold, R_lo er2, R_hi a0, R_hi b0, R_lo r0)
{
  Deltas interior, boundary;
  R_lo y = 2 * r0;
  R_lo x = y * w / h;
  for (N_16 i = 0; i < w; ++i)
  {
    R_lo u = x * ((i + 0.5) / w - 0.5);
    for (N_16 j = 0; j < h; ++j)
    {
      R_lo v = y * ((j + 0.5) / h - 0.5);
      ((i == 0 || j == 0 || i == w - 1 || j == h - 1) ? boundary : interior)
      [Coord(i,j)] = Delta(u, v, 0, 0);
    }
  }
  return RegionSeries(Orbit(a0,b0,maxiters,glitch_threshold,er2),0,ZeroSeries(order),ZeroSeries(order),interior,boundary);
}

/*
burningShip :: Precision p => Int -> Double -> Int -> Region p -> [[Region p]]
burningShip order e si = iterate (concat . parMap rseq (burningShipReg order e si)) . pure

escaped r x y = not $ x * x + y * y < r
```

```
smapPair f (a, b) = ((,) $! a) $! f b

*/

bool escaped(R_lo er2, R_lo x, R_lo y)
{
  return ! (x * x + y * y < er2);
}

bool escaped(R_lo er2, const Delta &d)
{
  return escaped(er2, d.x, d.y);
}

bool inaccurate(N_16 order, R_lo threshold, const RegionSeries &r)
{
  R_lo error = 0;
  for (auto m : r.boundary)
  {
    const Delta &d = m.second;
    const Delta &s = biSeries(order, r.s, r.t, d);
    R_lo dx = d.x - s.x;
    R_lo dy = d.y - s.y;
    R_lo sx = (d.x + s.x)/2;
    R_lo sy = (d.y + s.y)/2;
    R_lo q = (dx * dx + dy * dy) / (sx * sx + sy * sy);
    if (std::isnan(q) || std::isinf(q)) q = 0;
    error = std::max(error, q);
  }
  error = sqrt(error);
  return !(error < threshold);
}

/*
inaccurate e r = not . (e >) . sqrt . maximum . fmap err . boundary $ r
  where
    count = fromIntegral . M.size . boundary $ r
    err d =
      let d' = biSeries (s r, t r) d
          dx = x d - x d'
          dy = y d - y d'
          sx = (x d + x d') / 2
          sy = (y d + y d') / 2
      in  case (dx ^ 2 + dy ^ 2) / (sx ^ 2 + sy ^ 2) of
            q | isNaN q || isInfinite q -> 0
              | otherwise -> q
*/

/*
f <|$|> m = M.fromAscList . parMap rseq (smapPair f) . M.toAscList $ m
infixl 4 <|$|>

isFinished RegionFinished{} = True
isFinished _ = False

burningShipReg :: Precision p => Int -> Double -> Int -> Region p -> [Region p]
```

```
600
    burningShipReg _ _ _ RegionFinished{} = []

    burningShipReg _ _ _ r@(RegionSingleton{})
      | escaped er2 (x $ toDelta (ref r)) (y $ toDelta (ref r)) = [RegionFinished{ ↙
          ↳ done = M.singleton (ix r) (result (n (ref r)) (toDelta (ref r))) }]
605   | otherwise = [r{ ref = burningShipRef (ref r) }]

    */

    /**
610 std::vector<Region> burningShipReg(const RegionSingleton &r, R_lo er2)
    {
      std::vector<Region> rs;
      Delta dr(r.ref);
      if (escaped(er2, dr))
615   {
        RegionFinished r2;
        r2[r.ij] = result(r.ref.n, dr);
        rs.push_back(Region(r2));
      }
620   else
      {
        RegionSingleton r2(r);
        r2.ref = burningShipRef(r.ref);
        rs.push_back(Region(r2));
625   }
      return rs;
    }
    */

630 /*
    burningShipReg _ _ _ r@(RegionPerturb{})
      | escaped er2 xr0 yr0 = [RegionFinished{ done = const ReferenceEscaped <$> ↙
          ↳ active r }]
      | M.null active'' = [RegionFinished{ done = done' }]
      | otherwise = [RegionFinished{ done = done' }, r{ ref = ref', active = active↙
          ↳ '' }]
635   where
        ref' = burningShipRef (ref r)
        ref_ = toDelta (ref r)
        ref_1 = toDelta ref'
        xr0 = x ref_
640     yr0 = y ref_
        xr = x ref_1
        yr = y ref_1
        (newDone, active')    = M.partition (\d -> escaped er2 (x d + xr) (y d + yr)↙
            ↳ ) (burningShipDel ref_ <$> active r)
        (newGlitch, active'') = M.partition (\d -> xr^2 + yr^2 > 1000 * ((xr + x d)↙
            ↳ ^2 + (yr + y d)^2)) active'
645     done' = (result (n ref') <$> newDone) `M.union` (const PerturbationGlitch <$↙
            ↳ > newGlitch)
    */

    const double glitch_threshold = 0.001;

650 std::vector<Region> burningShipReg(const RegionPerturb &r, Z_64 maxiters, R_lo ↙
```

```
    ↪ er2)
{
  std::vector<Region> rs;
  if (escaped(er2, r.orbit.x[r.n], r.orbit.y[r.n]))
  {
std::cerr << "r.n " << r.n << std::endl;
    RegionFinished done;
    for (auto d : r.active)
    {
      done[d.first] = ReferenceEscaped;
    }
    rs.push_back(Region(done));
  }
  else
  {
    RegionFinished done;
//    Deltas active;
//    Reference ref2 = burningShipRef(r.ref);
//    Delta dr2(ref2);
//    R_lo gr2 = glitch_threshold * (dr2.x * dr2.x + dr2.y * dr2.y);
//    #pragma omp parallel for
    const Z_64 count = r.active.size();
std::cerr << "count " << count << std::endl;
    std::vector<Coord> ijs;
    std::vector<Delta> ds;
    ijs.reserve(count);
    ds.reserve(count);
    for (auto d : r.active)
    {
      ijs.push_back(d.first);
      ds.push_back(d.second);
    }
    const Z_64 osize = r.orbit.x.size();
    const Z_64 m = osize - 1;
std::cerr << "osize " << osize << std::endl;
std::cerr << "maxiters " << maxiters << std::endl;
    #pragma omp parallel for
    for (Z_64 ix = 0; ix < count; ++ix)
    {
      Coord ij = ijs[ix];
      Delta d = ds[ix];
      R_lo a = d.a;
      R_lo b = d.b;
      R_lo x = d.x;
      R_lo y = d.y;
      bool finished = false;
      for (Z_64 n = r.n; n < m; ++n)
      {
        R_lo X = r.orbit.x[n];
        R_lo Y = r.orbit.y[n];
        R_lo Xn = r.orbit.x[n+1];
        R_lo Yn = r.orbit.y[n+1];
        R_lo Gn = r.orbit.g[n+1];
        R_lo xn = (2 * X + x) * x - (2 * Y + y) * y + a;
        R_lo yn = 2 * diffabs(X * Y, X * y + x * Y + x * y) + b;
        R_lo xx = Xn + xn;
        R_lo yy = Yn + yn;
```

```
          R_lo z2 = xx * xx + yy * yy;
          x = xn;
          y = yn;
          if (! (z2 < er2))
          {
            #pragma omp critical
            done[ij] = result(n, xx, yy);
            finished = true;
            break;
          }
          else if (! (z2 > Gn))
          {
            #pragma omp critical
            done[ij] = PerturbationGlitch;
            finished = true;
            break;
          }
        }
        if (! finished)
        {
          #pragma omp critical
          done[ij] = osize == maxiters ? Unescaped : ReferenceEscaped;
        }
      }
      if (size(done) > 0)
        rs.push_back(Region(done));
/**
      if (active.size() > 0)
        rs.push_back(Region(RegionPerturb(ref2, active)));
*/
    }
    return rs;
}


/*
burningShipReg order e si r@(RegionSeries{})
  | snd (size r) == 1 = [RegionSingleton{ ref = ref r, ix = case M.keys (↙
      ↪ interior r) ++ M.keys (boundary r) of [ij] -> ij }]
  | escaped (er2) (x $ toDelta (ref r)) (y $ toDelta (ref r)) = [RegionFinished{↙
      ↪  done = M.fromList [ (ij, ReferenceEscaped) | ij <- M.keys (interior r) ↙
      ↪ ++ M.keys (boundary r)] }]
  | n (ref r) >= si = traceShow ("forcestop", M.size (interpolate r), n (ref r))↙
      ↪  $ [RegionPerturb{ ref = ref r, active = interpolate r }]
  | otherwise = case nub . map (burningShipFolding (toDelta $ ref r)) . M.elems ↙
      ↪ $ boundary r of
      [q] -> -- optimized singleton case
        let (s', t') = burningShipSeries order q (toDelta $ ref r) (s r, t r)
            r' = r{ ref = ref', s = s', t = t', boundary = bry' }
        in  if inaccurate e r' then traceShow ("inaccurate1", M.size (↙
            ↪ interpolate r), n (ref r)) $ [RegionPerturb{ ref = ref r, active =↙
            ↪  interpolate r }] else [r']
      qs -> map newRegion qs
  where
    ref' = burningShipRef (ref r)
    bry' = burningShipDel (toDelta $ ref r) <$> boundary r
    q0 = burningShipFolding (toDelta $ ref r) (Delta 0 0 0 0)
```

```
    newRegion q
      | q == q0 = let (s', t') = burningShipSeries order q (toDelta $ ref r) (s r, t r)
                      (bry, int) = M.partitionWithKey (\ix _ -> any (`M.notMember` region) (neighbours ix)) (burningShipDel (toDelta $ ref r) <$> region)
                      r' = r{ ref = ref', s = s', t = t', interior = int, boundary = bry }
                  in  if inaccurate e r' then traceShow ("inaccurateQ", M.size region, n (ref r)) $ RegionPerturb{ ref = ref r, active = region } else traceShow ("okQ", M.size region, n (ref r)) $ r'
      | otherwise =                                traceShow ("noncentral", M.size region, n (ref r)) $ RegionPerturb{ ref = ref r, active = region }
      where
        region = M.filter ((q ==) . burningShipFolding (toDelta $ ref r)) (interpolate r)
*/

std::vector<Region> burningShipReg(const RegionSeries &r, N_16 order, R_lo series_threshold, Z_64 series_iters, R_lo er2)
{
  std::vector<Region> rs;
/**
  if (size(r) == 1)
  {
    for (auto d : r.interior)
      rs.push_back(Region(RegionSingleton(r.orbit, r.n, d.first)));
    for (auto d : r.boundary)
      rs.push_back(Region(RegionSingleton(r.orbit, r.n, d.first)));
  }
  else
*/
  {
    Delta dr(0, 0, r.orbit.x[r.n], r.orbit.y[r.n]);
    if (escaped(er2, dr))
    {
      RegionFinished done;
      for (auto d : r.boundary)
        done[d.first] = ReferenceEscaped;
      for (auto d : r.interior)
        done[d.first] = ReferenceEscaped;
      rs.push_back(Region(done));
    }
    else if (r.n >= series_iters)
    {
      rs.push_back(Region(RegionPerturb(r.orbit, r.n, interpolate(order, r))));
    }
    else
    {
      Deltas newBoundary;
      std::set<Quadrant> qs;
      Quadrant q;
      for (auto d : r.boundary)
      {
        newBoundary[d.first] = burningShipDel(dr, d.second);
        qs.insert(q = burningShipFolding(dr, d.second));
```

```
        }

        if (qs.size() == 1)
        {
          auto st = burningShipSeries(order, q, dr, r.s, r.t);
          RegionSeries r2(r.orbit, r.n + 1, st.first, st.second, r.interior, ↙
              ↳ newBoundary);
          if (inaccurate(order, series_threshold, r2))
          {
            rs.push_back(Region(RegionPerturb(r.orbit, r.n, interpolate(order, r))↙
                ↳ ));
          }
          else
          {
            rs.push_back(Region(r2));
          }
        } else {
          Quadrant q0 = burningShipFolding(dr);
          for (auto q : qs)
          {
            Deltas region;
            for (auto ds : interpolate(order, r))
              if (q == burningShipFolding(dr, ds.second))
                region[ds.first] = ds.second;
            if (q == q0)
            {
              auto st = burningShipSeries(order, q, dr, r.s, r.t);
              Deltas boundary, interior;
              for (auto ds : region)
              {
                Delta d = burningShipDel(dr, ds.second);
                bool isEdge = false;
                for (auto ix : neighbours(ds.first))
                  if (! region.count(ix))
                    isEdge = true;
                (isEdge ? boundary : interior)[ds.first] = d;
              }
              RegionSeries r2(r.orbit, r.n + 1, st.first, st.second, interior, ↙
                  ↳ boundary);
              if (inaccurate(order, series_threshold, r2))
                rs.push_back(Region(RegionPerturb(r.orbit, r.n, region)));
              else
                rs.push_back(Region(r2));
            }
            else
            {
              rs.push_back(Region(RegionPerturb(r.orbit, r.n, region)));
            }
          }
        }
      }
      return rs;
    }

    std::vector<Region> burningShipReg(const Region &r, Z_64 maxiters, N_16 order, ↙
```

```
    ↪ R_lo series_threshold, Z_64 series_iters, R_lo er2, R_lo glitch_threshold)
{
  switch (r.tag)
  {
    case RegionTypeFinished: return std::vector<Region>();
/**
    case RegionTypeSingleton: return burningShipReg(r.Singleton.value(), er2);
*/
    case RegionTypePerturb: return burningShipReg(r.Perturb.value(), maxiters, ↙
        ↪ er2);
    case RegionTypeSeries: return burningShipReg(r.Series.value(), order, ↙
        ↪ series_threshold, series_iters, er2);
  }
  assert(!"burningShipReg Region.tag");
}

/*

--          region = M.filter ((q ==) . burningShipFolding ref_) deltas
{-
          -- compute center of region
          (Sum is, Sum js) = M.foldMapWithKey (\(i, j) _ -> (Sum (fromIntegral i↙
              ↪ ), Sum (fromIntegral j))) region
          m = fromIntegral (M.size region)
          i0 = is / m
          j0 = js / m
          -- move reference to center of region
          ij = findKeyNear (i0, j0) (M.keys region)
          d = burningShipDel ref_ $ region M.! ij
          aa' = aa ref' + realToFrac (a d)
          bb' = bb ref' + realToFrac (b d)
          xx' = xx ref' + realToFrac (x d)
          yy' = yy ref' + realToFrac (y d)
-}
{-
          -- move series to new reference
          (s', t') = burningShipSeries q ref_ (s r, t r)
--          s' = biShift (-a d) (-b d) $ s r
--          t' = biShift (-a d) (-b d) $ t r
          (bry, int) = M.partitionWithKey (\ix _ -> any (`M.notMember` region) (↙
              ↪ neighbours ix)) (({- (\d' -> Delta (a d' - a d) (b d' - b d) (x ↙
              ↪ d' - x d) (y d' - y d)) . -} burningShipDel ref_) <$> region)
      in  --traceShow (n ref', ix, M.size region, M.size int, M.size bry) $
          RegionSeries
            { ref = ref'--{ aa = aa', bb = bb', xx = xx', yy = yy' }
            , s = s'
            , t = t'
            , interior = int
            , boundary = bry
            }

-}

{-
uniShift c m = go (order - 1) m
  where
```

```
    go i m | i >= 0 = go (i - 1) (go2 i m)
           | otherwise = m
      where
        go2 j m | j < order = go2 (j + 1) $ M.insert j (m ! j + m ! (j + 1) * c)↙
            ↳ m
                | otherwise = m
    m ! j = case M.lookup j m of Nothing -> 0 ; Just x -> x

biShift x y
  = twiddle . pull . fmap (uniShift x) . push
  . twiddle . pull . fmap (uniShift y) . push

push :: (Ord a, Ord b) => Map (a, b) v -> Map a (Map b v)
push m = M.fromListWith M.union [ (i, M.singleton j e) | ((i,j), e) <- M.assocs ↙
    ↳ m ]

pull :: (Ord a, Ord b) => Map a (Map b v) -> Map (a, b) v
pull m = M.fromList [((i,j),e) | (i, mj) <- M.assocs m, (j, e) <- M.assocs mj ]

twiddle :: (Ord a, Ord b) => Map (a, b) v -> Map (b, a) v
twiddle = M.mapKeys (\(i,j) -> (j,i))
-}

{-
horner shift
for (i = n - 2; i >= 0; i--)
for (j = i; j < n - 1; j++)
fmpz_addmul(poly + j, poly + j + 1, c); // poly[j] += poly[j+1]*c

a_0 x^0 + a_1 x^1 + a_2 x^2 + a_3 x^3
->
a_2 += a_3 * c

a_1 += a_2 * c
a_2 += a_3 * c

a_0 += a_1 * c
a_1 += a_2 * c
a_2 += a_3 * c
-}


findKeyNear (i0, j0) = minimumBy (comparing d2)
  where d2 (i, j) = let x = fromIntegral i - i0 ; y = fromIntegral j - j0 in x *↙
      ↳ x + y * y
*/

/*
{-
substitute

x = sum_{i>=0,j>=0,i+j>0} S_{i,j} a^i b^j
y = sum_{i>=0,j>=0,i+j>0} T_{i,j} a^i b^j

into

x := 2 X x - 2 Y y + x^2 - y^2 + a
```

```
y := 2 * diffabs (X Y) (X y + x Y + x y) + b

diffabs :: (Ord a, Num a) => a -> a -> a
diffabs c d =
  if (c >= 0)
    then
      if (c + d >= 0)
        then d               -- y :=  2 * (       X y + x Y + x y) + b
        else -(2 * c + d)    -- y := -2 * (2 X Y + X y + x Y + x y) + b
    else
      if (c + d > 0)
        then 2 * c + d       -- y :=  2 * (2 X Y + X y + x Y + x y) + b
        else -d              -- y := -2 * (       X y + x Y + x y) + b


and equate coefficients gives iteration formulas for the series approximation
-}
*/

/*
main' :: Precision p => Int -> Int -> Int -> Int -> Double -> Int -> Rounded ↙
    ↪ TowardNearest p -> Rounded TowardNearest p -> Double -> proxy p -> [((Int,↙
    ↪  Int), Result)]
main' w h maxiters order threshold seriesiters a0 b0 r0 _prec
  = concatMap (M.toList . done)
  . filter isFinished
  . concat
  . takeWhile (not . null)
  . take maxiters
  . burningShip order threshold seriesiters
  $ initial w h order a0 b0 r0

plot1 w h maxiters u ((i, j), r) = let k = j * w + i in case r of
    ReferenceEscaped   -> U.write u k (-1/0)
    PerturbationGlitch -> U.write u k (-1/0)
    Result mu          -> U.write u k (mu)

clamp x lo hi = lo `max` x `min` hi

*/

void calculate(double *buffer, N_16 w, N_16 h, Z_64 maxiters, N_16 order, R_lo ↙
    ↪ series_threshold, Z_64 series_iters, R_lo glitch_threshold, R_lo er2, ↙
    ↪ const R_hi &a0, const R_hi &b0, R_lo r0)
{
  std::vector<Region> active;
  active.push_back(initial(w, h, maxiters, order, glitch_threshold, er2, a0, b0,↙
      ↪  r0));
  for (auto i = 0; i < maxiters; ++i)
  {
    if (active.empty()) break;
    std::vector<Region> next;
    for (auto source : active)
    {
      std::cerr << source.tag << std::endl;
      std::vector<Region> news = burningShipReg(source, maxiters, order, ↙
          ↪ series_threshold, series_iters, er2, glitch_threshold);
```

```
      for (auto sink : news)
      {
        if (sink.tag == RegionTypeFinished)
        {
          for (auto p : sink.Finished.value())
          {
            Z_64 i = p.first.first;
            Z_64 j = p.first.second;
            Z_64 k = j * w + i;
            buffer[k] = p.second;
          }
        }
        else
        {
          next.push_back(sink);
        }
      }
    }
    std::swap(active, next);
  }
}

/*

plot w h maxiters xs = runST (do
  u <- U.new (w * h)
  mapM_ (plot1 w h maxiters u) xs
  v <- S.new (w * h * 3)
  mapM_ (colour1 w h u v) [ (i,j) | i <- [0 .. w - 1], j <- [0 .. h - 1] ]
  unsafeFreeze v)

colour1 w h u v (i, j) = do
  let i1 = if i == 0 then i + 1 else i - 1
      j1 = if j == 0 then j + 1 else j - 1
  (n00) <- U.read u (j  * w + i )
  (n01) <- U.read u (j  * w + i1)
  (n10) <- U.read u (j1 * w + i )
  (n11) <- U.read u (j1 * w + i1)
  let du =  (n00 - n11)
      dv =  (n01 - n10)
      de = du^2 + dv^2
      wo :: Word8
      l = 20 * log de
      wo = floor l
      (r,g,b)
        | n00 == 0 = (0, 0, 0) -- unescaped
        | n00 == -1/0 = (255, 0, 0) -- glitch
        | isNaN de = (0, 0, 255) -- error?
        | otherwise = (wo, 128, floor n00)
      k = 3 * (j * w + i)
  S.write v (k + 0) r
  S.write v (k + 1) g
  S.write v (k + 2) b
*/

void colour(N_8 *image, const double *buffer, N_16 w, N_16 h)
{
```

```
  #pragma omp parallel for
  for (Z_64 j = 0; j < h; ++j)
  {
    Z_64 j1 = j == 0 ? j + 1 : j - 1;
    for (Z_64 i = 0; i < w; ++i)
    {
      Z_64 i1 = i == 0 ? i + 1 : i - 1;
      {
        auto n00 = buffer[j  * w + i ];
        auto n01 = buffer[j  * w + i1];
        auto n10 = buffer[j1 * w + i ];
        auto n11 = buffer[j1 * w + i1];
        auto du = n00 - n11;
        auto dv = n01 - n10;
        auto de = du * du + dv * dv;
        auto l = 128 + 20 * log(de);
        N_8 r = l;
        N_8 g = 128;
        N_8 b = n00;
        if (n00 == 0) { r = g = b = 0; }
        if (n00 == PerturbationGlitch) { r = 255; g = b = 0; }
        if (n00 == ReferenceEscaped) { r = b = 255; g = 0; }
        if (std::isnan(de)) { r = g = 0; b = 255; }
        Z_64 k = (j * w + i) * 3;
        image[k + 0] = r;
        image[k + 1] = g;
        image[k + 2] = b;
      }
    }
  }
}


/*
main :: IO ()
main = do
  [sw, sh, smaxiters, sorder, sthreshold, sseriesiters, sa0, sb0, sr0] <- ↙
      ↳ getArgs
  let w = read sw
      h = read sh
{-
      a0s = "-1.765724070777579936114404613125203747548323196202122799408000137"
      b0s = "-0.041893714510242382733641617814193925623948756785781053295999999"
      r0 = 2 / 9.9035203142830852e27
      maxiters = 200100
-}
{-
  let a0 = -1.7750157188827608561390422879089556558379945626926644647 68
      b0 = -0.046245056219157170454983287250662144618728655141566663527
      r0 = 2 / 1.2980742146337048E33
      maxiters = 5100
-}
      maxiters = read smaxiters
      order = read sorder
      threshold = read sthreshold
      seriesiters = read sseriesiters
      r0 = read sr0
```

```
      prec = ceiling $ 24 - logBase 2 r0
      img = plot w h maxiters $ reifyPrecision prec $ main' w h maxiters order ↙
          ↳ threshold seriesiters (read sa0) (read sb0) r0
  hPutStr stderr $ "precision bits: " ++ show prec ++ "\n"
  hPutStr stdout $ "P6\n" ++ show w ++ " " ++ show h ++ "\n255\n"
  unsafeWith img (\p -> hPutBuf stdout p (w * h * 3))
*/

int main(int argc, char **argv)
{
  if (argc != 12)
  {
    fprintf(stderr, "usage: %s width height iterations series_order ↙
        ↳ series_threshold series_iterations glitch_threshold escape_radius a0 ↙
        ↳ b0 r0\n", argv[0]);
    return 1;
  }
  N_16 w = std::atoi(argv[1]);
  N_16 h = std::atoi(argv[2]);
  Z_64 maxiters = std::atol(argv[3]);
  N_16 order = std::atoi(argv[4]);
  R_lo series_threshold = std::strtold(argv[5], nullptr);
  Z_64 series_iters = std::atol(argv[6]);
  R_lo glitch_threshold = std::strtold(argv[7], nullptr);
  R_lo escape_radius = std::strtold(argv[8], nullptr);
  R_lo er2 = escape_radius * escape_radius;
  R_lo r0 = std::strtold(argv[11], nullptr);
  Z_64 prec = std::ceil(24 - std::log2(r0));
  R_hi::set_default_prec(prec);
  R_hi a0(argv[9]);
  R_hi b0(argv[10]);
  Z_64 bytes = size_t(w) * size_t(h);
  double *buffer = (double *) std::malloc(bytes * sizeof(double));
  calculate(buffer, w, h, maxiters, order, series_threshold, series_iters, ↙
      ↳ glitch_threshold, er2, a0, b0, r0);
  N_8 *image = (N_8 *) std::malloc(bytes * 3);
  colour(image, buffer, w, h);
  std::free(buffer);
  std::printf("P6\n%d %d\n255\n", int(w), int(h));
  std::fwrite(image, bytes * 3, 1, stdout);
  std::free(image);
std::cerr << "\t" << w << std::endl;
std::cerr << "\t" << h << std::endl;
std::cerr << "\t" << maxiters << std::endl;
std::cerr << "\t" << order << std::endl;
std::cerr << "\t" << series_threshold << std::endl;
std::cerr << "\t" << series_iters << std::endl;
std::cerr << "\t" << glitch_threshold << std::endl;
std::cerr << "\t" << escape_radius << std::endl;
std::cerr << "\t" << a0 << std::endl;
std::cerr << "\t" << b0 << std::endl;
std::cerr << "\t" << r0 << std::endl;
  return 0;
}
```

18 burning-ship-series-approximation/BossaNova.hs

```
{-# LANGUAGE DataKinds #-}
{-# LANGUAGE FlexibleContexts #-}

--import Control.Concurrent.Spawn (parMapIO_)
import Control.Monad (join)
import Control.Monad.ST (runST)
import Control.Parallel.Strategies (parMap, rseq)
import Data.List (minimumBy, nub, transpose)
import Data.Map.Strict (Map)
import qualified Data.Map.Strict as M
import Data.Monoid (Sum(..))
import Data.Ord (comparing)
import Data.Vector.Storable (unsafeFreeze, unsafeWith)
import qualified Data.Vector.Storable.Mutable as S
import qualified Data.Vector.Unboxed.Mutable  as U
import Data.Word (Word8)
import Numeric.Rounded (Rounded, RoundingMode(TowardNearest), Precision, ↙
    ↳ reifyPrecision)
import System.Environment (getArgs)
import System.IO (hPutBuf, hPutStr, stdout, stderr)
import Debug.Trace (trace, traceShow)

diffabs :: (Ord a, Num a) => a -> a -> a
diffabs c d =
  if (c >= 0)
    then
      if (c + d >= 0)
        then d
        else -(2 * c + d)
    else
      if (c + d > 0)
        then 2 * c + d
        else -d

er :: Double
er = 65536

er2 :: Double
er2 = er^2

type REAL p = Rounded TowardNearest p

data Delta = Delta{ a, b, x, y :: !Double }

data Region p = RegionSingleton{ ref :: !(Reference p), ix :: !(Int, Int) }
              | RegionSeries{ ref :: !(Reference p), s, t :: !(BiSeries), interior↙
                  ↳ , boundary :: !(Map (Int, Int) Delta) }
              | RegionPerturb{ ref :: !(Reference p), active :: !(Map (Int, Int) ↙
                  ↳ Delta) }
              | RegionFinished{ done :: !(Map (Int, Int) Result) }

size :: Region p -> (Char, Int)
size r@(RegionSingleton{}) = ('1', 1)
size r@(RegionSeries{}) = ('S', M.size (interior r) + M.size (boundary r))
size r@(RegionPerturb{}) = ('P', M.size (active r))
size r@(RegionFinished{}) = ('F', M.size (done r))
```

```
55  data Result = ReferenceEscaped | PerturbationGlitch | Result Double deriving ↙
        ↳ Show

    initial :: Precision p => Int -> Int -> Int -> REAL p -> REAL p -> Double -> ↙
        ↳ Region p
    initial w h order a0 b0 r0 = RegionSeries
      { ref = Reference a0 b0 0 0 0
60    , s = zero order
      , t = zero order
      , interior = int
      , boundary = bry
      }
65    where
        (bry, int) = M.partitionWithKey (\(i, j) _ -> i == 0 || j == 0 || i == w - 1↙
            ↳  || j == h - 1) deltas
        deltas = M.fromList
          [ ((i, j), Delta u v 0 0)
          | i <- [0 .. w - 1]
70        , j <- [0 .. h - 1]
          , let u = 2 * r0 * ((fromIntegral i + 0.5) / fromIntegral w - 0.5) * ↙
              ↳ fromIntegral w / fromIntegral h
          , let v = 2 * r0 * ((fromIntegral j + 0.5) / fromIntegral h - 0.5)
          ]

75  burningShip :: Precision p => Int -> Double -> Int -> Region p -> [[Region p]]
    burningShip order e si = iterate (concat . parMap rseq (burningShipReg order e ↙
        ↳ si)) . pure

    escaped r x y = not $ x * x + y * y < r

80  smapPair f (a, b) = ((,) $! a) $! f b

    interpolate r = (biSeries (s r, t r) <$> interior r) `M.union` boundary r

    inaccurate e r = not . (e >) . sqrt . maximum . fmap err . boundary $ r
85    where
        count = fromIntegral . M.size . boundary $ r
        err d =
          let d' = biSeries (s r, t r) d
              dx = x d - x d'
90            dy = y d - y d'
              sx = (x d + x d') / 2
              sy = (y d + y d') / 2
          in  case (dx ^ 2 + dy ^ 2) / (sx ^ 2 + sy ^ 2) of
                q | isNaN q -> 0
95                  | otherwise -> q

    result m d = Result $ fromIntegral m + 1 - log (log (sqrt (x d ^ 2 + y d ^ 2))) ↙
        ↳ / log 2

    f <|$|> m = M.fromAscList . parMap rseq (smapPair f) . M.toAscList $ m
100 infixl 4 <|$|>

    isFinished RegionFinished{} = True
    isFinished _ = False

105 burningShipReg :: Precision p => Int -> Double -> Int -> Region p -> [Region p]
```

```
burningShipReg _ _ _ RegionFinished{} = []

burningShipReg _ _ _ r@(RegionSingleton{})
  | escaped er2 (x $ toDelta (ref r)) (y $ toDelta (ref r)) = [RegionFinished{ done = M.singleton (ix r) (result (n (ref r)) (toDelta (ref r))) }]
  | otherwise = [r{ ref = burningShipRef (ref r) }]

burningShipReg _ _ _ r@(RegionPerturb{})
  | escaped er2 xr0 yr0 = [RegionFinished{ done = const ReferenceEscaped <$> active r }]
  | M.null active'' = [RegionFinished{ done = done' }]
  | otherwise = [RegionFinished{ done = done' }, r{ ref = ref', active = active'' }]
  where
    ref' = burningShipRef (ref r)
    ref_ = toDelta (ref r)
    ref_1 = toDelta ref'
    xr0 = x ref_
    yr0 = y ref_
    xr = x ref_1
    yr = y ref_1
    (newDone, active')    = M.partition (\d -> escaped er2 (x d + xr) (y d + yr)) (burningShipDel ref_ <$> active r)
    (newGlitch, active'') = M.partition (\d -> xr^2 + yr^2 > 1000 * ((xr + x d)^2 + (yr + y d)^2)) active'
    done' = (result (n ref') <$> newDone) `M.union` (const PerturbationGlitch <$> newGlitch)

burningShipReg order e si r@(RegionSeries{})
  | snd (size r) == 1 = [RegionSingleton{ ref = ref r, ix = case M.keys (interior r) ++ M.keys (boundary r) of [ij] -> ij }]
  | escaped (er2) (x $ toDelta (ref r)) (y $ toDelta (ref r)) = [RegionFinished{ done = M.fromList [ (ij, ReferenceEscaped) | ij <- M.keys (interior r) ++ M.keys (boundary r)] }]
  | n (ref r) >= si = traceShow ("forcestop", M.size (interpolate r), n (ref r)) $ [RegionPerturb{ ref = ref r, active = interpolate r }]
  | otherwise = case nub . map (burningShipFolding (toDelta $ ref r)) . M.elems $ boundary r of
      [q] -> -- optimized singleton case
        let (s', t') = burningShipSeries order q (toDelta $ ref r) (s r, t r)
            r' = r{ ref = ref', s = s', t = t', boundary = bry' }
        in  if inaccurate e r' then traceShow ("inaccurate1", M.size (interpolate r), n (ref r)) $ [RegionPerturb{ ref = ref r, active = interpolate r }] else [r']
      qs -> map newRegion qs
  where
    ref' = burningShipRef (ref r)
    bry' = burningShipDel (toDelta $ ref r) <$> boundary r
    q0 = burningShipFolding (toDelta $ ref r) (Delta 0 0 0 0)
    newRegion q
{-  | q == q0 -}= let (s', t') = burningShipSeries order q (toDelta $ ref r) (s r, t r)
                      (bry, int) = M.partitionWithKey (\ix _ -> any (`M.notMember` region) (neighbours ix)) (burningShipDel (toDelta $ ref r) <$> region)
                      r' = r{ ref = ref', s = s', t = t', interior = int,
```

```
                        ↳ boundary = bry }
                in  if inaccurate e r' || M.null int then traceShow ("↙
                    ↳ inaccurateQ", M.size region, n (ref r)) $ RegionPerturb{↙
                    ↳  ref = ref r, active = region } else traceShow ("okQ", M↙
                    ↳ .size region, n (ref r)) $ r'
--      | otherwise =                                traceShow ("noncentral", M.size ↙
    ↳ region, n (ref r)) $ RegionPerturb{ ref = ref r, active = region }
      where
        region = M.filter ((q ==) . burningShipFolding (toDelta $ ref r)) (↙
            ↳ interpolate r)
--          region = M.filter ((q ==) . burningShipFolding ref_) deltas
{-
          -- compute center of region
          (Sum is, Sum js) = M.foldMapWithKey (\(i, j) _ -> (Sum (fromIntegral i↙
              ↳ ), Sum (fromIntegral j))) region
          m = fromIntegral (M.size region)
          i0 = is / m
          j0 = js / m
          -- move reference to center of region
          ij = findKeyNear (i0, j0) (M.keys region)
          d = burningShipDel ref_ $ region M.! ij
          aa' = aa ref' + realToFrac (a d)
          bb' = bb ref' + realToFrac (b d)
          xx' = xx ref' + realToFrac (x d)
          yy' = yy ref' + realToFrac (y d)
-}
{-
          -- move series to new reference
          (s', t') = burningShipSeries q ref_ (s r, t r)
--          s' = biShift (-a d) (-b d) $ s r
--          t' = biShift (-a d) (-b d) $ t r
          (bry, int) = M.partitionWithKey (\ix _ -> any (`M.notMember` region) (↙
              ↳ neighbours ix)) (({- (\d' -> Delta (a d' - a d) (b d' - b d) (x ↙
              ↳ d' - x d) (y d' - y d)) . -} burningShipDel ref_) <$> region)
      in  --traceShow (n ref', ix, M.size region, M.size int, M.size bry) $
          RegionSeries
            { ref = ref'--{ aa = aa', bb = bb', xx = xx', yy = yy' }
            , s = s'
            , t = t'
            , interior = int
            , boundary = bry
            }

-}

{-
uniShift c m = go (order - 1) m
  where
    go i m | i >= 0 = go (i - 1) (go2 i m)
           | otherwise = m
      where
        go2 j m | j < order = go2 (j + 1) $ M.insert j (m ! j + m ! (j + 1) * c)↙
            ↳ m
                | otherwise = m
    m ! j = case M.lookup j m of Nothing -> 0 ; Just x -> x

biShift x y
```

```
  = twiddle . pull . fmap (uniShift x) . push
195  . twiddle . pull . fmap (uniShift y) . push

push :: (Ord a, Ord b) => Map (a, b) v -> Map a (Map b v)
push m = M.fromListWith M.union [ (i, M.singleton j e) | ((i,j), e) <- M.assocs ↙
    ↳ m ]

200 pull :: (Ord a, Ord b) => Map a (Map b v) -> Map (a, b) v
pull m = M.fromList [((i,j),e) | (i, mj) <- M.assocs m, (j, e) <- M.assocs mj ]

twiddle :: (Ord a, Ord b) => Map (a, b) v -> Map (b, a) v
twiddle = M.mapKeys (\(i,j) -> (j,i))
205 -}

{-
horner shift
for (i = n - 2; i >= 0; i--)
210 for (j = i; j < n - 1; j++)
fmpz_addmul(poly + j, poly + j + 1, c); // poly[j] += poly[j+1]*c

a_0 x^0 + a_1 x^1 + a_2 x^2 + a_3 x^3
->
215 a_2 += a_3 * c

a_1 += a_2 * c
a_2 += a_3 * c

220 a_0 += a_1 * c
a_1 += a_2 * c
a_2 += a_3 * c
-}

225

neighbours (i, j) = [ (i - 1, j), (i + 1, j), (i, j - 1), (i, j + 1) ]
findKeyNear (i0, j0) = minimumBy (comparing d2)
  where d2 (i, j) = let x = fromIntegral i - i0 ; y = fromIntegral j - j0 in x *↙
      ↳  x + y * y

230 data Reference p = Reference{ aa, bb, xx, yy :: !(REAL p), n :: !Int }

toDelta :: Precision p => Reference p -> Delta
toDelta r = Delta
  { a = realToFrac (aa r)
235  , b = realToFrac (bb r)
  , x = realToFrac (xx r)
  , y = realToFrac (yy r)
  }

240 burningShipRef :: Precision p => Reference p -> Reference p
burningShipRef r = r
  { xx = xx r ^ 2 - yy r ^ 2 + aa r
  , yy = abs (2 * xx r * yy r) + bb r
  , n = n r + 1
245  }

burningShipDel :: Delta -> Delta -> Delta
burningShipDel r d = d
```

```
  { x = 2 * x r * x d - 2 * y r * y d + x d ^ 2 - y d ^ 2 + a d
  , y = 2 * diffabs (x r * y r) (x r * y d + x d * y r + x d * y d) + b d
  }

burningShipFolding :: Delta -> Delta -> Int
burningShipFolding r d = diffabsIx (x r * y r) (x r * y d + x d * y r + x d * y ↙
    ↳ d)
{-
  | x r + x d >= 0 && y r + y d >= 0 = 0
  | x r + x d <  0 && y r + y d >= 0 = 1
  | x r + x d <  0 && y r + y d <  0 = 2
  | x r + x d >= 0 && y r + y d <  0 = 3
  | otherwise = traceShow (x r, x d, x r + x d, y r, y d, y r + y d) 0
-}

type BiSeries = Map (Int, Int) Double

burningShipSeries order q r (s, t) = (s', t')
  where
    s' = M.fromList [((i,j),
      (if (i,j) == (1,0) then 1 else 0) +
      2 * xr * s M.! (i,j) - 2 * yr * t M.! (i,j) +
      sum [ s M.! (k, l) * s M.! (i - k, j - l)
          - t M.! (k, l) * t M.! (i - k, j - l)
          | k <- [0..i], l <- [0..j] ])
      | i <- [0..order], j <- [0..order], i + j <= order ]
    t' = M.fromList [((i,j),
      (if (i,j) == (0,1) then 1 else 0) +
      case q of
        0 ->  2 * (xr * t M.! (i, j) + s M.! (i, j) * yr + sum [ s M.! (k, l) * ↙
            ↳ t M.! (i - k, j - l) | k <- [0..i], l <- [0..j] ])
        1 -> -2 * (xr * t M.! (i, j) + s M.! (i, j) * yr + sum [ s M.! (k, l) * ↙
            ↳ t M.! (i - k, j - l) | k <- [0..i], l <- [0..j] ] + if (i,j) == ↙
            ↳ (0,0) then 2 * xr * yr else 0)
        2 ->  2 * (xr * t M.! (i, j) + s M.! (i, j) * yr + sum [ s M.! (k, l) * ↙
            ↳ t M.! (i - k, j - l) | k <- [0..i], l <- [0..j] ] + if (i,j) == ↙
            ↳ (0,0) then 2 * xr * yr else 0)
        3 -> -2 * (xr * t M.! (i, j) + s M.! (i, j) * yr + sum [ s M.! (k, l) * ↙
            ↳ t M.! (i - k, j - l) | k <- [0..i], l <- [0..j] ])
      )
      | i <- [0..order], j <- [0..order], i + j <= order ]
    xr = x r
    yr = y r

zero order = M.fromList [((i,j), 0) | i <- [0..order], j <- [0..order], i + j <=↙
    ↳  order ]

biSeries (s, t) d = d{ x = biSeries1 s (ad) (bd), y = biSeries1 t (ad) (bd) }
  where ad = a d
        bd = b d
biSeries1 s a b = sum [ e * a^i * b^j | ((i, j), e) <- M.assocs s ]

{-
substitute

x = sum_{i>=0,j>=0,i+j>0} S_{i,j} a^i b^j
y = sum_{i>=0,j>=0,i+j>0} T_{i,j} a^i b^j
```

```
    into
300
    x := 2 X x - 2 Y y + x^2 - y^2 + a
    y := 2 * diffabs (X Y) (X y + x Y + x y) + b

    diffabs :: (Ord a, Num a) => a -> a -> a
305 diffabs c d =
      if (c >= 0)
        then
          if (c + d >= 0)
            then d                -- y :=  2 * (       X y + x Y + x y) + b
310         else -(2 * c + d)     -- y := -2 * (2 X Y + X y + x Y + x y) + b
        else
          if (c + d > 0)
            then 2 * c + d        -- y :=  2 * (2 X Y + X y + x Y + x y) + b
            else -d               -- y := -2 * (       X y + x Y + x y) + b
315

    and equate coefficients gives iteration formulas for the series approximation
    -}

320 diffabsIx :: (Ord a, Num a) => a -> a -> Int
    diffabsIx c d =
      if (c >= 0)
        then
          if (c + d >= 0)
325         then 0
            else 1
        else
          if (c + d > 0)
            then 2
330         else 3

    main' :: Precision p => Int -> Int -> Int -> Int -> Double -> Int -> Rounded ↙
      ↳ TowardNearest p -> Rounded TowardNearest p -> Double -> proxy p -> [((Int,↙
      ↳  Int), Result)]
    main' w h maxiters order threshold seriesiters a0 b0 r0 _prec
      = concatMap (M.toList . done)
335   . filter isFinished
      . concat
      . takeWhile (not . null)
      . take maxiters
      . burningShip order threshold seriesiters
340   $ initial w h order a0 b0 r0

    plot1 w h maxiters u ((i, j), r) = let k = j * w + i in case r of
        ReferenceEscaped   -> U.write u k (-1/0)
        PerturbationGlitch -> U.write u k (-1/0)
345     Result mu          -> U.write u k (mu)

    colour1 w h u v (i, j) = do
      let i1 = if i == 0 then i + 1 else i - 1
          j1 = if j == 0 then j + 1 else j - 1
350   (n00) <- U.read u (j  * w + i )
      (n01) <- U.read u (j  * w + i1)
      (n10) <- U.read u (j1 * w + i )
```

```
  (n11) <- U.read u (j1 * w + i1)
  let du =  (n00 - n11)
      dv =  (n01 - n10)
      de = du^2 + dv^2
      wo :: Word8
      l = 20 * log de
      wo = floor l
      (r,g,b)
        | n00 == 0 = (0, 0, 0) -- unescaped
        | n00 == -1/0 = (255, 0, 0) -- glitch
        | isNaN de = (0, 0, 255) -- error?
        | otherwise = (wo, 128, floor n00)
      k = 3 * (j * w + i)
  S.write v (k + 0) r
  S.write v (k + 1) g
  S.write v (k + 2) b

clamp x lo hi = lo `max` x `min` hi

plot w h maxiters xs = runST (do
  u <- U.new (w * h)
  mapM_ (plot1 w h maxiters u) xs
  v <- S.new (w * h * 3)
  mapM_ (colour1 w h u v) [ (i,j) | i <- [0 .. w - 1], j <- [0 .. h - 1] ]
  unsafeFreeze v)

main :: IO ()
main = do
  [sw, sh, smaxiters, sorder, sthreshold, sseriesiters, sa0, sb0, sr0] <- ↙
      ↳ getArgs
  let w = read sw
      h = read sh
{-
      a0s = "-1.765724070777579936114404613125203747548323196202122799408000137"
      b0s = "-0.041893714510242382733641617814193925623948756785781053295999999"
      r0 = 2 / 9.9035203142830852e27
      maxiters = 200100
-}
{-
  let a0 = -1.77501571888276085613904228790895565583799456269266446476 8
      b0 = -0.046245056219157170454983287250662144618728655141566663527
      r0 = 2 / 1.2980742146337048E33
      maxiters = 5100
-}
      maxiters = read smaxiters
      order = read sorder
      threshold = read sthreshold
      seriesiters = read sseriesiters
      r0 = read sr0
      prec = ceiling $ 24 - logBase 2 r0
      img = plot w h maxiters $ reifyPrecision prec $ main' w h maxiters order ↙
          ↳ threshold seriesiters (read sa0) (read sb0) r0
  hPutStr stderr $ "precision bits: " ++ show prec ++ "\n"
  hPutStr stdout $ "P6\n" ++ show w ++ " " ++ show h ++ "\n255\n"
  unsafeWith img (\p -> hPutBuf stdout p (w * h * 3))
```

19 burning-ship-series-approximation/.gitignore

```
BossaNova
BossaNova2
*.hi
*.o
*.hp
*.svg
*.kfr
```

20 burning-ship-series-approximation/Makefile

```
BossaNova2: BossaNova2.cxx
	g++ -std=c++17 -Wall -Wextra -pedantic -Ofast -march=native -mtune=native -mcpu=native -fopenmp -o BossaNova2 BossaNova2.cxx -lmpfr -lgmp -lm
```

21 .gitignore

```
mandelbrot-delta-cl/mandelbrot-delta-cl
pjulia-c
pjulia-hs
pjulia.o
pjulia.hi
*.gif
*.pgm
*.ppm
*.png
*.dat
*.log
mandelbrot-series-approximation/m
mandelbrot-series-approximation/m.html
mandelbrot-series-approximation/m.html.mem
mandelbrot-series-approximation/m.js
mandelbrot-series-approximation/worker.js
mandelbrot-series-approximation/worker.js.mem
buddhabrot/bb
buddhabrot/bbrender
buddhabrot/bound
buddhabrot/bound-2
buddhabrot/cusp
buddhabrot/tip
buddhabrot/histogram
buddhabrot/bb.map
buddhabrot/bbcolourizelayers
buddhabrot/bbrenderlayers
buddhabrot/bs.map
buddhabrot/bscolourizelayers
buddhabrot/bsrenderlayers
julia-iim/julia-de
julia-iim/julia-iim
julia-iim/julia-lsm
julia-iim/julia-iim-rainbow
z-to-exp-z-plus-c/z-to-exp-z-plus-c
z-to-exp-z-plus-c/z-to-exp-z-plus-c-henriksen
z-to-exp-z-plus-c/z-to-exp-z-plus-c-lyapunov
```

22 julia/.gitignore

```
j-render
```

23 julia-iim/julia-de.c

```
// gcc -std=c99 -Wall -Wextra -pedantic -O3 -o julia-de julia-de.c -lm

#include <complex.h>
#include <math.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>

void hsv2rgb(double h, double s, double v, int *rp, int *gp, int *bp) {
  double i, f, p, q, t, r, g, b;
  int ii;
  if (s == 0.0) { r = g = b = v; } else {
    h = 6 * (h - floor(h));
    ii = i = floor(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;
    }
  }
  *rp = fmin(fmax(round(r * 255), 0), 255);
  *gp = fmin(fmax(round(g * 255), 0), 255);
  *bp = fmin(fmax(round(b * 255), 0), 255);
}

complex double julia_attractor(complex double c, int maxiters, int *period) {
  double epsilon = nextafter(2, 4) - 2;
  complex double z = c;
  double mzp = 1.0/0.0;
  int p = 0;
  for (int n = 1; n < maxiters; ++n) {
    double mzn = cabs(z);
    if (mzn < mzp) {
      mzp = mzn;
      p = n;
      complex double z0 = z;
      for (int i = 0; i < 64; ++i) {
        complex double f = z0;
        complex double df = 1;
        for (int j = 0; j < p; ++j) {
          df = 2 * f * df;
          f = f * f + c;
        }
```

```
        complex double z1 = z0 - (f - z0) / (df - 1);
        if (cabs(z1 - z0) <= epsilon) {
          z0 = z1;
          break;
        }
        if (isinf(creal(z1)) || isinf(cimag(z1)) || isnan(creal(z1)) || isnan(↙
            ↳ cimag(z1))) {
          break;
        }
        z0 = z1;
      }
      complex double w = z0;
      complex double dw = 1;
      for (int i = 0; i < p; ++i) {
        dw = 2 * w * dw;
        w = w * w + c;
      }
      if (cabs(dw) <= 1) {
        *period = p;
        return z0;
      }
    }
    z = z * z + c;
  }
  *period = 0;
  return 0;
}

double julia_exterior_de(complex double c, complex double z, int maxiters, ↙
    ↳ double escape_radius) {
  complex double dz = 1;
  for (int n = 0; n < maxiters; ++n) {
    if (cabs(z) >= escape_radius) {
      return cabs(z) * log(cabs(z)) / cabs(dz);
    }
    dz = 2 * z * dz;
    z = z * z + c;
  }
  return 0;
}

double julia_interior_de(complex double c, complex double z, int maxiters, ↙
    ↳ double escape_radius, double pixel_size, complex double z0, int period, ↙
    ↳ bool superattracting, int *fatou) {
  complex double dz = 1;
  for (int n = 0; n < maxiters; ++n) {
    if (cabs(z) >= escape_radius) {
      *fatou = -1;
      return cabs(z) * log(cabs(z)) / cabs(dz);
    }
    if (cabs(z - z0) <= pixel_size * pixel_size) {
//      fprintf(stderr, "z-z0 = %.18f + %.18fi\tdz = %.18f + %.18f i\n", creal(z↙
    ↳ -z0), cimag(z-z0), creal(dz), cimag(dz));
      *fatou = n % period;
      if (superattracting) {
        return cabs(z - z0) * fabs(log(cabs(z - z0))) / cabs(dz);
      } else {
```

```
        return cabs(z - z0) / cabs(dz);
      }
    }
    dz = 2 * z * dz;
    z = z * z + c;
  }
  *fatou = -2;
  return 0;
}

int main(int argc, char **argv) {
  int size = 512;
  double radius = 2;
  double escape_radius = 1 << 10;
  int maxiters = 1 << 24;
  if (! (argc > 2)) { return 1; }
  complex double c = atof(argv[1]) + I * atof(argv[2]);

  int period = 0;
  bool superattracting = false;
  complex double z0 = julia_attractor(c, maxiters, &period);
  if (period > 0) {
    double epsilon = nextafter(1, 2) - 1;
    complex double z = z0;
    complex double dz = 1;
    for (int i = 0; i < period; ++i) {
      dz = 2 * z * dz;
      z = z * z + c;
    }
    superattracting = cabs(dz) <= epsilon;
    fprintf(stderr, "z0 = %.18f + %.18f i\ndz = %.18f + %.18f i\nperiod = %d\
        ↪ nsuperattracting = %d\n", creal(z0), cimag(z0), creal(dz), cimag(dz), \
        ↪ period, superattracting);
  }

  double pixel_size = 2 * radius / size;
  printf("P6\n%d %d\n255\n", size, size);
  for (int j = 0; j < size; ++j) {
    for (int i = 0; i < size; ++i) {
      double x = 2 * radius * ((i + 0.5) / size - 0.5);
      double y = 2 * radius * (0.5 - (j + 0.5) / size);
      complex double z = x + I * y;
      double de = 0;
      int fatou = -1;
      if (period > 0) {
        de = julia_interior_de(c, z, maxiters, escape_radius, pixel_size, z0, \
            ↪ period, superattracting, &fatou);
      } else {
        de = julia_exterior_de(c, z, maxiters, escape_radius);
      }
      int r, g, b;
      hsv2rgb(fatou / (double) period, 0.25 * (0 <= fatou), tanh(de / pixel_size\
          ↪ ), &r, &g, &b);
      putchar(r);
      putchar(g);
      putchar(b);
    }
```

```
  }
  return 0;
}
```

24 julia-iim/julia-iim.c

```
// gcc -std=c99 -Wall -Wextra -pedantic -O3 -ffast-math -o julia-iim julia-iim.c
    ↪ -lm
// ./julia-iim > j.pgm

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

#define dpi 300
#define inches 6

#define size (dpi * inches)
unsigned char pgm[size][size];

int main()
{
  memset(&pgm[0][0], 255, size * size);
  double _Complex z = 1;
  double scale = size / 3.0;
  for (int i = 0; i < 1 << 29; ++i)
  {
    int x = floor((creal(z) + 1.0) * scale);
    int y = floor((cimag(z) + 1.5) * scale);
    pgm[y][x] = 0;
    int coin = rand() > RAND_MAX/2;
    if (coin)
      z = 0.5 + csqrt(0.25 - z);
    else
      z = 0.5 - csqrt(0.25 - z);
  }
  fprintf(stdout, "P5\n%d %d\n255\n", size, size);
  fwrite(&pgm[0][0], size * size, 1, stdout);
  return 0;
}
```

25 julia-iim/julia-iim-rainbow.c

```
// gcc -std=c99 -Wall -Wextra -pedantic -O3 -ffast-math -o julia-iim-rainbow
    ↪ julia-iim-rainbow.c -lm
// ./julia-iim-rainbow -0.125000000000000 0.649519052838329 > fat-rabbit.ppm

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

#define dpi 300
```

```
#define inches 6

#define size (dpi * inches)
unsigned char ppm[size][size][3];

static const double pi = 3.141592653589793;

double cnorm(double _Complex z)
{
  double x = creal(z);
  double y = cimag(z);
  return x * x + y * y;
}

inline void hsv2rgb(double hue, double sat, double bri, int *r, int *g, int *b)
{
  hue -= floor(hue);
  hue *= 6;
  int i = (int) floor(hue);
  double f = hue - i;
  if (! (i & 1))
    f = 1.0 - f;
  double m = bri * (1.0 - sat);
  double n = bri * (1.0 - sat * f);
  switch (i)
  {
  default:
  case 0: *r = 255 * bri;
    *g = 255 * n;
    *b = 255 * m;
    break;
  case 1: *r = 255 * n;
    *g = 255 * bri;
    *b = 255 * m;
    break;
  case 2: *r = 255 * m;
    *g = 255 * bri;
    *b = 255 * n;
    break;
  case 3: *r = 255 * m;
    *g = 255 * n;
    *b = 255 * bri;
    break;
  case 4: *r = 255 * n;
    *g = 255 * m;
    *b = 255 * bri;
    break;
  case 5: *r = 255 * bri;
    *g = 255 * m;
    *b = 255 * n;
  }
}

struct node
{
  struct node *next;
  double _Complex z;
```

```
  double h;
};

static struct node *freelist = 0;
struct node *allocate()
{
  if (freelist)
  {
    struct node *n = freelist;
    freelist = freelist->next;
    n->next = 0;
    n->z = 0;
    n->h = 0;
    return n;
  }
  else
  {
    return calloc(1, sizeof(struct node));
  }
}
void deallocate(struct node *n)
{
  n->next = freelist;
  freelist = n;
}

int main(int argc, char **argv)
{
  double _Complex c;
  if (argc > 2)
    c = atof(argv[1]) + I * atof(argv[2]);
  else if (argc > 1)
    c = atof(argv[1]);
  else
    c = 0;
  // repelling fixed point
  memset(&ppm[0][0][0], 0, size * size * 3);
  double scale = size / 5.0;
  double small = (0.25 * 0.25) / (scale * scale);
  double big = (1.0 * 1.0) / (scale * scale);
  // initialize loop
  struct node *loop = 0;
  for (int i = 0; i < 1 << 4; ++i)
  {
    double h = (i + 0.5) / (1 << 4);
    struct node *n = allocate();
    n->next = loop;
    n->z = 4 * cexp(2 * pi * I * h);
    n->h = h;
    loop = n;
  }
  // iterate
  for (int j = 0; j < 1 << 8; ++j)
  {
    // step loop twice (two branches)
    struct node *new_loop = 0;
    double _Complex z1 = 4;
```

```
125       for (int pass = 0; pass < 2; ++pass)
          {
            struct node *old_loop = loop;
            while (old_loop)
            {
130           // pick the root nearest the previous point in the new loop
              double _Complex z = csqrt(old_loop->z - c);
              double d1 = cnorm(z1 - z);
              double d2 = cnorm(z1 + z);
              struct node *n = allocate();
135           n->next = new_loop;
              n->z = d1 < d2 ? z : -z;
              n->h = (old_loop->h + pass) * 0.5;
              new_loop = n;
              struct node *next = old_loop->next;
140           if (pass == 1)
                deallocate(old_loop);
              old_loop = next;
              z1 = n->z;
            }
145       }
          struct node *old_loop = new_loop;
          new_loop = 0;
          double _Complex z0 = old_loop->z;
          double _Complex h0 = old_loop->h;
150       old_loop = old_loop->next;
          while (old_loop)
          {
            if (cnorm(old_loop->z - z0) < small)
            {
155           // prune points that are too closely spaced
              struct node *next = old_loop->next;
              deallocate(old_loop);
              old_loop = next;
            }
160         else if (cnorm(old_loop->z - z0) < big)
            {
              // move points that are reasonably spaced from old to new loop
              z0 = old_loop->z;
              h0 = old_loop->h;
165           struct node *next = old_loop->next;
              old_loop->next = new_loop;
              new_loop = old_loop;
              old_loop = next;
            }
170         else
            {
              // interpolate points that too distantly spaced
              struct node *next = allocate();
              next->next = old_loop;
175           next->z = (z0 + old_loop->z) * 0.5;
              next->h = (h0 + old_loop->h) * 0.5;
              old_loop = next;
            }
          }
180       loop = new_loop;
        }
```

```
  // plot loop
  while (loop)
  {
    int x = floor((creal(loop->z) + 2.5) * scale);
    int y = floor((cimag(loop->z) + 2.5) * scale);
    int r, g, b;
    hsv2rgb(loop->h, 1, 1, &r, &g, &b);
    ppm[y][x][0] = r;
    ppm[y][x][1] = g;
    ppm[y][x][2] = b;
    struct node *next = loop->next;
    deallocate(loop);
    loop = next;
  }
  fprintf(stdout, "P6\n# Julia set for %.18f + %.18f i\n%d %d\n255\n", creal(c),↙
      ↪ cimag(c), size, size);
  fwrite(&ppm[0][0][0], size * size * 3, 1, stdout);
  return 0;
}
```

26 julia-iim/julia-lsm.c

```
// gcc -std=c99 -Wall -Wextra -pedantic -O3 -ffast-math -o julia-lsm julia-lsm.c↙
    ↪ -lm -fopenmp
// ./julia-lsm > j.ppm

#include <complex.h>
#include <math.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define aa 1
#define dpi 256
#define inches 6

#define size (aa * dpi * inches)
int status[size][size];
unsigned char ppm[size][size][3];

int main()
{
  double r2 = (3.0 / size) * (3.0 / size);
  {
    double _Complex z = 0.5;
    for (int k = 0; k < 1 << 24; ++k)
    {
      z = z - z * z;
      double z2 = creal(z) * creal(z) + cimag(z) * cimag(z);
      if (z2 < r2)
      {
        r2 = z2;
        break;
      }
    }
  }
```

```
35    #pragma omp parallel for
      for (int j = 0; j < size; ++j)
        for (int i = 0; i < size; ++i)
        {
          status[j][i] = 0;
40        double _Complex z = ((i + 0.0) * 3.0 / size - 1) + I * ((j + 0.0) * 3.0 / ↙
              ↳ size - 1.5);
          for (int k = 0; k < 1 << 24; ++k)
          {
            z = z - z * z;
            double z2 = creal(z) * creal(z) + cimag(z) * cimag(z);
45          if (z2 > 600)
            {
              status[j][i] = cimag(z) > 0 ? (k + 1) : (k + 1 + (1 << 24));
              break;
            }
50          if (z2 <= r2)
            {
              status[j][i] = -(k + 1);
              break;
            }
55        }
        }
      memset(&ppm[0][0][0], 255, size * size * 3);
      #pragma omp parallel for
      for (int j = 0; j < size; ++j)
60    {
        int jj = j;
        int jj1 = j-1;
        if (jj1 < 0) jj1 = 0;
        for (int i = 0; i < size; ++i)
65      {
          int ii = i;
          int ii1 = i-1;
          if (ii1 < 0) ii1 = 0;
          bool level = status[jj][ii] != status[jj1][ii1] || status[jj][ii1] != ↙
              ↳ status[jj1][ii];
70        bool julia = (status[jj][ii]>0) != (status[jj1][ii1]>0) || (status[jj][ii1↙
              ↳ ]>0) != (status[jj1][ii]>0);
          if (level)
          {
            if (status[jj][ii] > 0)
            {
75            ppm[j][i][1] *= 0;
              ppm[j][i][2] *= 0;
            }
            else
            {
80            ppm[j][i][0] *= 0;
              ppm[j][i][2] *= 0;
            }
          }
          if (julia)
85  #if 1
            for (int j2 = j - 2; j2 <= j + 2; ++j2)
              for (int i2 = i - 2; i2 <= i + 2; ++i2)
              {
```

```
            double r2 = (i2-i) * (i2-i) + (j2-j) * (j2-j);
            if (r2 <= 4)
            {
#else
            {
              int j2 = j;
              int i2 = i;
#endif
              ppm[j2][i2][0] *= 0;
              ppm[j2][i2][1] *= 0;
              ppm[j2][i2][2] *= 0;
            }
#if 1
          }
#endif
      }
    }
    fprintf(stdout, "P6\n%d %d\n255\n", size, size);
    fwrite(&ppm[0][0][0], size * size * 3, 1, stdout);
    return 0;
}
```

27 julia-iim/Makefile

```
julia-iim: julia-iim.c
        gcc -std=c99 -Wall -Wextra -pedantic -O3 -ffast-math -o julia-iim julia-↙
            ↳ iim.c -lm

julia-lsm: julia-lsm.c
        gcc -std=c99 -Wall -Wextra -pedantic -O3 -ffast-math -o julia-lsm julia-↙
            ↳ lsm.c -lm -fopenmp

julia-de: julia-de.c
        gcc -std=c99 -Wall -Wextra -pedantic -O3 -o julia-de julia-de.c -lm
```

28 julia/j-render.c

```
// gcc -std=c99 -Wall -Wextra -pedantic -O3 -fopenmp -o j-render j-render.c -↙
    ↳ lGLEW -lglfw -lGL -lm
// ./j-render out.ppm creal cimag log2size

#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include <complex.h>
#include <math.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>

#define GLSL(s) "#version 400 core\n" #s

const char *julia_vert_src = GLSL(
  layout(location = 0) in vec4 p;
  smooth out vec2 z0;
```

```
  void main() {
    z0 = p.zw;
    gl_Position = vec4(p.xy, 0.0, 1.0);
  }
);

const char *julia_frag_src = GLSL(
  uniform int maxiters;
  uniform int period;
  uniform float eps2;
  uniform float er2;
  uniform vec2 c;
  uniform vec2 w;
  smooth in vec2 z0;
  layout(location = 0) out float julia;
  vec2 csqr(vec2 z) {
    return vec2(z.x * z.x - z.y * z.y, 2.0 * z.x * z.y);
  }
  vec2 cmul(vec2 a, vec2 b) {
    return vec2(a.x * b.x - a.y * b.y, a.x * b.y + a.y * b.x);
  }
  void main() {
    vec2 z = z0;
    vec2 dz = vec2(0.5 * (length(dFdx(z0)) + length(dFdy(z0))), 0.0);
    float fatou = -3.0;
    for (int n = 0; n < min(maxiters, 65536); ++n) {
      float z2 = dot(z, z);
      if (z2 > er2) {
        float de = 0.5 * sqrt(z2) * log(z2) / length(dz);
        if (de < 4) {
          fatou = -2.0 + tanh(de);
        } else {
          fatou = -1.0;
        }
        break;
      }
      if (period > 0) {
        vec2 zw = z - w;
        float zw2 = dot(zw, zw);
        if (zw2 < eps2) {
          fatou = float(n % max(1, period));
          break;
        }
      }
      dz = 2.0 * cmul(z, dz);
      z = csqr(z) + c;
    }
    julia = fatou;
  }
);

const char *post_vert_src = GLSL(
  layout(location = 0) in vec4 p;
  smooth out vec2 uv;
  void main() {
    uv = p.zw;
    gl_Position = vec4(p.xy, 0.0, 1.0);
```

```
  }
);

/*

  a   b
    x
  c   d

a' = case a of
  -2 -> -2
  <= -1 -> -1
  _ -> a
ceil(a)

x
  a' <= -2                          -> 0 (iteration count exceeded)
  a' >= -1 && d' >= -1 && a' != d'  -> 0 (escaped to different components)
  c' >= -1 && b' >= -1 && c' != b'  -> 0 (escaped to different components)
  a' == b' == c' == d'   == -1      -> pow(product(all + 2), 0.25)  (geometric mean of de)
                         >  -1      -> 1 (escaped to same component)
  _                                 -> 1 (huh?)
*/

const char *post_frag_src = GLSL(
  uniform sampler2D t;
  smooth in vec2 uv;
  layout(location = 0) out float colour;
  void main() {
    ivec2 ij = ivec2(floor(uv));
    float a = texelFetch(t, ij, 0).x;
    float b = ij.x + 1 < textureSize(t, 0).x ? texelFetch(t, ij + ivec2(1, 0), 0).x : a;
    float c = ij.y + 1 < textureSize(t, 0).y ? texelFetch(t, ij + ivec2(0, 1), 0).x : a;
    float d = ij.x + 1 < textureSize(t, 0).x && ij.y + 1 < textureSize(t, 0).y ? texelFetch(t, ij + ivec2(1, 1), 0).x : a;
    float aa = ceil(a);
    float bb = ceil(b);
    float cc = ceil(c);
    float dd = ceil(d);
    float edge = 1.0;
    if (aa <= -2.0) { edge = 0.0; } else
    if (aa >= -1.0 && dd >= -1.0 && aa != dd) { edge = 0.0; } else
    if (bb >= -1.0 && cc >= -1.0 && bb != cc) { edge = 0.0; } else
    if (aa == bb && aa == cc && aa == dd && aa == -1.0) {
      edge = pow((a + 2.0) * (b + 2.0) * (c + 2.0) * (d + 2.0), 0.25);
    }
    colour = edge;
  }
);

const char *dim1_vert_src = GLSL(
  layout(location = 0) in vec4 p;
  smooth out vec2 uv;
  void main() {
```

```
    uv = p.zw;
    gl_Position = vec4(p.xy, 0.25, 1.0);
  }
);

const char *dim1_frag_src = GLSL(
  uniform sampler2D t;
  uniform float threshold;
  smooth in vec2 uv;
  layout(location = 0) out float colour;
  void main() {
    float value = texture(t, uv).x;
    if (value <= threshold) {
      discard;
    }
    colour = value;
  }
);

const char *dim2_vert_src = GLSL(
  layout(location = 0) in vec4 p;
  void main() {
    gl_Position = vec4(p.xy, 0.75, 1.0);
  }
);

const char *dim2_frag_src = GLSL(
  layout(location = 0) out float colour;
  void main() {
    colour = 0.0;
  }
);

void debug_program(GLuint program, const char *name) {
  if (program) {
    GLint linked = GL_FALSE;
    glGetProgramiv(program, GL_LINK_STATUS, &linked);
    if (linked != GL_TRUE) {
      printf("%s: OpenGL shader program link failed\n", name);
    }
    GLint length = 0;
    glGetProgramiv(program, GL_INFO_LOG_LENGTH, &length);
    char *buffer = (char *) malloc(length + 1);
    glGetProgramInfoLog(program, length, NULL, buffer);
    buffer[length] = 0;
    if (buffer[0]) {
      printf("%s: OpenGL shader program info log\n", name);
      printf("%s\n", buffer);
    }
    free(buffer);
  } else {
    printf("%s: OpenGL shader program creation failed\n", name);
  }
}

void debug_shader(GLuint shader, GLenum type, const char *name) {
  const char *tname = 0;
```

```
  switch (type) {
    case GL_VERTEX_SHADER:   tname = "vertex";   break;
    case GL_FRAGMENT_SHADER: tname = "fragment"; break;
    default:                 tname = "unknown";  break;
  }
  if (shader) {
    GLint compiled = GL_FALSE;
    glGetShaderiv(shader, GL_COMPILE_STATUS, &compiled);
    if (compiled != GL_TRUE) {
      printf("%s: OpenGL %s shader compile failed\n", name, tname);
    }
    GLint length = 0;
    glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &length);
    char *buffer = (char *) malloc(length + 1);
    glGetShaderInfoLog(shader, length, NULL, buffer);
    buffer[length] = 0;
    if (buffer[0]) {
      printf("%s: OpenGL %s shader info log\n", name, tname);
      printf("%s\n", buffer);
    }
    free(buffer);
  } else {
    printf("%s: OpenGL %s shader creation failed\n", name, tname);
  }
}

void compile_shader(GLint program, GLenum type, const char *name, const GLchar *↙
    ↳ source) {
  GLuint shader = glCreateShader(type);
  glShaderSource(shader, 1, &source, NULL);
  glCompileShader(shader);
  debug_shader(shader, type, name);
  glAttachShader(program, shader);
  glDeleteShader(shader);
}

GLint compile_program(const char *name, const GLchar *vert, const GLchar *frag) ↙
    ↳ {
  GLint program = glCreateProgram();
  if (vert) { compile_shader(program, GL_VERTEX_SHADER  , name, vert); }
  if (frag) { compile_shader(program, GL_FRAGMENT_SHADER, name, frag); }
  glLinkProgram(program);
  debug_program(program, name);
  return program;
}

complex double julia_attractor(complex double c, int maxiters, int *period) {
  double epsilon = nextafter(2, 4) - 2;
  complex double z = c;
  double mzp = 1.0/0.0;
  int p = 0;
  for (int n = 1; n < maxiters; ++n) {
    double mzn = cabs(z);
    if (mzn > 1024) {
      break;
    }
    if (mzn < mzp) {
```

```
      mzp = mzn;
      p = n;
      complex double z0 = z;
      for (int i = 0; i < 64; ++i) {
        complex double f = z0;
        complex double df = 1;
        for (int j = 0; j < p; ++j) {
          df = 2 * f * df;
          f = f * f + c;
        }
        complex double z1 = z0 - (f - z0) / (df - 1);
        if (cabs(z1 - z0) <= epsilon) {
          z0 = z1;
          break;
        }
        if (isinf(creal(z1)) || isinf(cimag(z1)) || isnan(creal(z1)) || isnan(↙
            ↳ cimag(z1))) {
          break;
        }
        z0 = z1;
      }
      complex double w = z0;
      complex double dw = 1;
      for (int i = 0; i < p; ++i) {
        dw = 2 * w * dw;
        w = w * w + c;
      }
      if (cabs(dw) <= 1) {
        *period = p;
        return z0;
      }
    }
    z = z * z + c;
  }
  *period = 0;
  return 0;
}

double simple_linear_regression(const double *ys, int n) {
  double xbar = 0;
  double ybar = 0;
  for (int x = 0; x < n; ++x) {
    xbar += x;
    ybar += ys[x];
  }
  xbar /= n;
  ybar /= n;
  double num = 0;
  double den = 0;
  for (int x = 0; x < n; ++x) {
    double dx = x - xbar;
    double dy = ys[x] - ybar;
    num += dx * dy;
    den += dx * dx;
  }
  return num / den;
}
```

```
void hsv2rgb(double h, double s, double v, int *red, int *grn, int *blu) {
  double i, f, p, q, t, r, g, b;
  int ii;
  if (s == 0.0) { r = g = b = v; } else {
    h = 6 * (h - floor(h));
    ii = i = floor(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 = fmin(fmax(255 * r + 0.5, 0), 255);
  *grn = fmin(fmax(255 * g + 0.5, 0), 255);
  *blu = fmin(fmax(255 * b + 0.5, 0), 255);
}

int main(int argc, char **argv) {
  if (argc != 5) {
    fprintf(stderr, "usage: %s out.ppm creal cimag log2size\n", argv[0]);
    return 1;
  }
  int wsize = 512;
  bool julia_save = true;
  const char *julia_filename = argv[1];
  complex double c = atof(argv[2]) - I * atof(argv[3]);
  int level = atoi(argv[4]) + 1;

  int tsize = 1 << level;
  int isize = tsize >> 1;

  float *julia_fatou = 0;
  float *julia_edge = 0;
  unsigned char *julia_ppm = 0;
  if (julia_save) {
    julia_fatou = malloc(tsize * tsize * sizeof(float));
    julia_edge = malloc(isize * isize * sizeof(float));
    julia_ppm = malloc(3 * isize * isize);
  }

  if (! glfwInit()) { return 1; }
  glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
  glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 0);
  glfwWindowHint(GLFW_OPENGL_FORWARD_COMPAT, GL_TRUE);
  glfwWindowHint(GLFW_OPENGL_PROFILE, GLFW_OPENGL_CORE_PROFILE);
  glfwWindowHint(GLFW_RESIZABLE, GL_FALSE);
  GLFWwindow *window = glfwCreateWindow(wsize, wsize, "j-render", 0, 0);
  if (! window) { glfwTerminate(); return 1; }
  glfwMakeContextCurrent(window);
```

```
  glewExperimental = GL_TRUE;
355 if (GLEW_OK != glewInit()) {
    return 1;
  }
  glGetError();

360 GLint julia_prog = compile_program("julia", julia_vert_src, julia_frag_src);
  GLuint julia_maxiters = glGetUniformLocation(julia_prog, "maxiters");
  GLuint julia_period = glGetUniformLocation(julia_prog, "period");
  GLuint julia_eps2 = glGetUniformLocation(julia_prog, "eps2");
  GLuint julia_er2 = glGetUniformLocation(julia_prog, "er2");
365 GLuint julia_c = glGetUniformLocation(julia_prog, "c");
  GLuint julia_w = glGetUniformLocation(julia_prog, "w");

  GLint post_prog = compile_program("post", post_vert_src, post_frag_src);
  GLuint post_t = glGetUniformLocation(post_prog, "t");
370
  GLint dim1_prog = compile_program("dim1", dim1_vert_src, dim1_frag_src);
  GLuint dim1_t = glGetUniformLocation(dim1_prog, "t");
  GLuint dim1_threshold = glGetUniformLocation(dim1_prog, "threshold");

375 GLint dim2_prog = compile_program("dim2", dim2_vert_src, dim2_frag_src);

  GLuint tex[3];
  glGenTextures(3, tex);
  glActiveTexture(GL_TEXTURE0);
380 glBindTexture(GL_TEXTURE_2D, tex[0]);
  glTexImage2D(GL_TEXTURE_2D, 0, GL_R32F, tsize, tsize, 0, GL_RED, GL_FLOAT, 0);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
385 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
  glActiveTexture(GL_TEXTURE1);
  glBindTexture(GL_TEXTURE_2D, tex[1]);
  glTexImage2D(GL_TEXTURE_2D, 0, GL_R32F, tsize, tsize, 0, GL_RED, GL_FLOAT, 0);
  glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
390 glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR_MIPMAP_LINEAR)↙
    ↳ ;
  glGenerateMipmap(GL_TEXTURE_2D);
  glActiveTexture(GL_TEXTURE2);
  glBindTexture(GL_TEXTURE_2D, tex[2]);
  glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT, tsize, tsize, 0, ↙
    ↳ GL_DEPTH_COMPONENT, GL_FLOAT, 0);
395
  GLenum bufs[1] = { GL_COLOR_ATTACHMENT0 };
  GLenum bufs2[2] = { GL_COLOR_ATTACHMENT0, GL_DEPTH_ATTACHMENT };
  GLuint fbo[3];
  glGenFramebuffers(3, fbo);
400 glBindFramebuffer(GL_DRAW_FRAMEBUFFER, fbo[0]);
  glFramebufferTexture2D(GL_DRAW_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, ↙
    ↳ GL_TEXTURE_2D, tex[0], 0);
  glDrawBuffers(1, bufs);
  glBindFramebuffer(GL_DRAW_FRAMEBUFFER, fbo[1]);
  glFramebufferTexture2D(GL_DRAW_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, ↙
    ↳ GL_TEXTURE_2D, tex[1], 0);
405 glDrawBuffers(1, bufs);
  glBindFramebuffer(GL_DRAW_FRAMEBUFFER, fbo[2]);
```

```
    glFramebufferTexture2D(GL_DRAW_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, ↙
        ↳ GL_TEXTURE_2D, tex[0], 0);
    glFramebufferTexture2D(GL_DRAW_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D↙
        ↳ , tex[2], 0);
    glDrawBuffers(2, bufs2);

    GLuint vbo;
    glGenBuffers(1, &vbo);
    glBindBuffer(GL_ARRAY_BUFFER, vbo);
    GLfloat r = 2.0;
    GLfloat vbo_data[3 * 4 * 4] =
      { -1, -1, -r, -r
      , -1,  1, -r,  r
      ,  1, -1,  r, -r
      ,  1,  1,  r,  r
      , -1, -1,  0, 0
      , -1,  1,  0, tsize
      ,  1, -1,  tsize, 0
      ,  1,  1,  tsize, tsize
      , -1, -1,  0, 0
      , -1,  1,  0, 1
      ,  1, -1,  1, 0
      ,  1,  1,  1, 1
      };
    glBufferData(GL_ARRAY_BUFFER, 3 * 4 * 4 * sizeof(GLfloat), vbo_data, ↙
        ↳ GL_STATIC_DRAW);

    GLuint julia_vao;
    glGenVertexArrays(1, &julia_vao);
    glBindVertexArray(julia_vao);
    glVertexAttribPointer(0, 4, GL_FLOAT, 0, 0, ((char *) 0) + 0 * 4 * 4 * sizeof(↙
        ↳ GLfloat));
    glEnableVertexAttribArray(0);

    GLuint post_vao;
    glGenVertexArrays(1, &post_vao);
    glBindVertexArray(post_vao);
    glVertexAttribPointer(0, 4, GL_FLOAT, 0, 0, ((char *) 0) + 1 * 4 * 4 * sizeof(↙
        ↳ GLfloat));
    glEnableVertexAttribArray(0);

    GLuint dim_vao;
    glGenVertexArrays(1, &dim_vao);
    glBindVertexArray(dim_vao);
    glVertexAttribPointer(0, 4, GL_FLOAT, 0, 0, ((char *) 0) + 2 * 4 * 4 * sizeof(↙
        ↳ GLfloat));
    glEnableVertexAttribArray(0);

    GLuint query[level + 1];
    glGenQueries(level + 1, query);

    glDisable(GL_DEPTH_TEST);

    glGetError();
    {
        int period = 0;
        complex double z0 = julia_attractor(c, 65536, &period);
```

```
        complex double z = z0;
        complex double dz = 1;
460     if (period) {
          for (int n = 0; n < period; ++n) {
            dz = 2 * z * dz;
            z = z * z + c;
          }
465     }

        glViewport(0, 0, tsize, tsize);

        glUseProgram(julia_prog);
470     glUniform1i(julia_maxiters, 65536);
        glUniform1i(julia_period, period);
        glUniform1f(julia_eps2, 5.0 / tsize);
        glUniform1f(julia_er2, 1024 * 1024);
        glUniform2f(julia_c, creal(c), cimag(c));
475     glUniform2f(julia_w, creal(z0), cimag(z0));
        glBindVertexArray(julia_vao);
        glBindFramebuffer(GL_DRAW_FRAMEBUFFER, fbo[0]);
        glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);

480     if (julia_save) {
          glActiveTexture(GL_TEXTURE0);
          glGetTexImage(GL_TEXTURE_2D, 0, GL_RED, GL_FLOAT, julia_fatou);
        }

485     glUseProgram(post_prog);
        glBindVertexArray(post_vao);
        glUniform1i(post_t, 0);
        glBindFramebuffer(GL_DRAW_FRAMEBUFFER, fbo[1]);
        glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
490     glActiveTexture(GL_TEXTURE1);
        glGenerateMipmap(GL_TEXTURE_2D);

        if (julia_save) {
          glGetTexImage(GL_TEXTURE_2D, 1, GL_RED, GL_FLOAT, julia_edge);
495       #pragma omp parallel for
          for (int jj = 0; jj < isize; ++jj) {
            for (int ii = 0; ii < isize; ++ii) {
              int fatou = ceilf(julia_fatou[(jj << 1) * tsize + (ii << 1)]);
              bool f = 0 <= fatou && fatou < period;
500           double hue = f ? fatou / (double) period : 0;
              double sat = f / 3.0;
              double val = julia_edge[jj * isize + ii];
              int red, grn, blu;
              hsv2rgb(hue, sat, val, &red, &grn, &blu);
505           int kk = jj * isize + ii;
              julia_ppm[3 * kk + 0] = red;
              julia_ppm[3 * kk + 1] = grn;
              julia_ppm[3 * kk + 2] = blu;
            }
510       }
          FILE *julia_file = fopen(julia_filename, "wb");
          fprintf(julia_file, "P6\n%d %d\n255\n", isize, isize);
          fwrite(julia_ppm, 3 * isize * isize, 1, julia_file);
          fclose(julia_file);
```

```
515         }

            // http://stackoverflow.com/questions/99906/count-image-similarity-on-gpu-↙
                ↳ opengl-occlusionquery
            glEnable(GL_DEPTH_TEST);
            glDepthFunc(GL_LEQUAL);
520         glBindFramebuffer(GL_DRAW_FRAMEBUFFER, fbo[2]);
            glBindVertexArray(dim_vao);
            for (int l = level; l >= 0; --l) { // 11's threshold is less than 1.0; 12'↙
                ↳ s equals 1.0
              glViewport(0, 0, 1 << l, 1 << l);
              glClear(GL_DEPTH_BUFFER_BIT);
525           glUseProgram(dim1_prog);
              glUniform1i(dim1_t, 1);
              glUniform1f(dim1_threshold, 1.0 - 0.5 * pow(0.25, level - l));
              glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
              glUseProgram(dim2_prog);
530           glBeginQuery(GL_SAMPLES_PASSED, query[l]);
              glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
              glEndQuery(GL_SAMPLES_PASSED);
            }
            glDisable(GL_DEPTH_TEST);
535         double logcount[level + 1];
            for (int l = level; l >= 0; --l) {
              GLuint count = 0;
              glGetQueryObjectuiv(query[l], GL_QUERY_RESULT, &count);
              logcount[l] = log2(count);
540         }
            double dim = fabs(simple_linear_regression(logcount + (level - 3), 3));
            printf("dim = %f\n", dim);

            GLuint e = glGetError();
545         if (e) {
              fprintf(stderr, "\nOpenGL error %d\n", e);
            }

          }
550
          glfwTerminate();
          return 0;
        }
```

29 mandelbrot-delta-cl/cl-extra.cc

```
const char *error_string(int err) {
  switch (err) {
  case CL_SUCCESS:                            return 0;
  case CL_DEVICE_NOT_FOUND:                   return "Device not found.";
5 case CL_DEVICE_NOT_AVAILABLE:               return "Device not available";
  case CL_COMPILER_NOT_AVAILABLE:             return "Compiler not available";
  case CL_MEM_OBJECT_ALLOCATION_FAILURE:      return "Memory object allocation ↙
      ↳ failure";
  case CL_OUT_OF_RESOURCES:                   return "Out of resources";
  case CL_OUT_OF_HOST_MEMORY:                 return "Out of host memory";
10 case CL_PROFILING_INFO_NOT_AVAILABLE:      return "Profiling information not ↙
      ↳ available";
  case CL_MEM_COPY_OVERLAP:                   return "Memory copy overlap";
```

```
    case CL_IMAGE_FORMAT_MISMATCH:               return "Image format mismatch";
    case CL_IMAGE_FORMAT_NOT_SUPPORTED:          return "Image format not supported
        ↪ ";
    case CL_BUILD_PROGRAM_FAILURE:               return "Program build failure";
    case CL_MAP_FAILURE:                         return "Map failure";
    case CL_INVALID_VALUE:                       return "Invalid value";
    case CL_INVALID_DEVICE_TYPE:                 return "Invalid device type";
    case CL_INVALID_PLATFORM:                    return "Invalid platform";
    case CL_INVALID_DEVICE:                      return "Invalid device";
    case CL_INVALID_CONTEXT:                     return "Invalid context";
    case CL_INVALID_QUEUE_PROPERTIES:            return "Invalid queue properties";
    case CL_INVALID_COMMAND_QUEUE:               return "Invalid command queue";
    case CL_INVALID_HOST_PTR:                    return "Invalid host pointer";
    case CL_INVALID_MEM_OBJECT:                  return "Invalid memory object";
    case CL_INVALID_IMAGE_FORMAT_DESCRIPTOR:     return "Invalid image format
        ↪ descriptor";
    case CL_INVALID_IMAGE_SIZE:                  return "Invalid image size";
    case CL_INVALID_SAMPLER:                     return "Invalid sampler";
    case CL_INVALID_BINARY:                      return "Invalid binary";
    case CL_INVALID_BUILD_OPTIONS:               return "Invalid build options";
    case CL_INVALID_PROGRAM:                     return "Invalid program";
    case CL_INVALID_PROGRAM_EXECUTABLE:          return "Invalid program executable
        ↪ ";
    case CL_INVALID_KERNEL_NAME:                 return "Invalid kernel name";
    case CL_INVALID_KERNEL_DEFINITION:           return "Invalid kernel definition
        ↪ ";
    case CL_INVALID_KERNEL:                      return "Invalid kernel";
    case CL_INVALID_ARG_INDEX:                   return "Invalid argument index";
    case CL_INVALID_ARG_VALUE:                   return "Invalid argument value";
    case CL_INVALID_ARG_SIZE:                    return "Invalid argument size";
    case CL_INVALID_KERNEL_ARGS:                 return "Invalid kernel arguments";
    case CL_INVALID_WORK_DIMENSION:              return "Invalid work dimension";
    case CL_INVALID_WORK_GROUP_SIZE:             return "Invalid work group size";
    case CL_INVALID_WORK_ITEM_SIZE:              return "Invalid work item size";
    case CL_INVALID_GLOBAL_OFFSET:               return "Invalid global offset";
    case CL_INVALID_EVENT_WAIT_LIST:             return "Invalid event wait list";
    case CL_INVALID_EVENT:                       return "Invalid event";
    case CL_INVALID_OPERATION:                   return "Invalid operation";
    case CL_INVALID_GL_OBJECT:                   return "Invalid OpenGL object";
    case CL_INVALID_BUFFER_SIZE:                 return "Invalid buffer size";
    case CL_INVALID_MIP_LEVEL:                   return "Invalid mip-map level";
    default: return "Unknown";
    }
}

void error_print(int err, int loc) {
  if (err == CL_SUCCESS) { return; }
  fprintf(stderr, "CL ERROR: %d %s (%d)\n", err, error_string(err), loc);
  exit(1);
}

#define E(err) error_print(err, __LINE__)
```

30 mandelbrot-delta-cl/delta.cl

```
__kernel void prefixsum
  ( __global const int *input
```

```
  , __global int *output
  , const int count
  , const int passbit
  ) {
  int i = get_global_id(0);
  if (0 <= i && i < count) {
    int x = input[i];
    if (i & passbit) {
      int j = (i | (passbit - 1)) - passbit;
      if (0 <= j && j < count) {
        x += input[j];
      }
    }
    output[i] = x;
  }
}

__kernel void permute
  ( __global const double *input
  , __global const int *keep
  , __global const int *index
  , __global double *output
  , const int count
  , const int elements
  ) {
  int i = get_global_id(0);
  if (0 <= i && i < count) {
    int k = keep[i];
    if (k) {
      int j = index[i] - 1;
      if (0 <= j && j < count) {
        int ii = i * elements;
        int jj = j * elements;
        for (int k = 0; k < elements; ++k) {
          output[jj + k] = input[ii + k];
        }
      }
    }
  }
}

__kernel void initialize
  ( __global double *output
  ,          const int      width
  ,          const int      height
  , __global const double *reference
  ,          const double  deltaX
  ,          const double  deltaY
  ) {
  int count = width * height;
  int i = get_global_id(0);
  if (0 <= i && i < count) {
    double x = (deltaX + (i % width) - (width /2))/(height/2);
    double y = (deltaY + (i / width) - (height/2))/(height/2);
    double xn = x;
    double yn = y;
    double dx = 0.0;
```

```
      double dy = 0.0;
      double ex = 0.0;
      double ey = 0.0;
      int k = 0;
      for (int j = 0; j < 12; ++j) {
        double cx = reference[k++];
        double cy = reference[k++];
        dx += cx * xn - cy * yn;
        dy += cx * yn + cy * xn;
        cx = reference[k++];
        cy = reference[k++];
        ex += cx * xn - cy * yn;
        ey += cx * yn + cy * xn;
        double xn1 = x * xn - y * yn;
        double yn1 = x * yn + y * xn;
        xn = xn1;
        yn = yn1;
      }
      const int elements = 5;
      int jj = i * elements;
      output[jj + 0] = i;
      output[jj + 1] = dx;
      output[jj + 2] = dy;
      output[jj + 3] = ex;
      output[jj + 4] = ey;
    }
  }

  __kernel void iterate
    ( __global const double *reference // { zx, zy, ax, ay }
    , __global const double *input     // { idx, dx, dy, lx, ly }
    , __global        double *output    // { idx, dx, dy, lx, ly }
    , __global int           *keep      // { !escaped }
    , __global float         *image     // { dwell, angle, distance, atomperiod }
    ,           const int     count
    ,           const int     width
    ,           const int     height
    ,           const double deltaX
    ,           const double deltaY
    ,           const double pixelSpacing
    ,           const int     start
    ,           const int     iters
    ,           const int     period
    ) {
    int i = get_global_id(0);
    if (0 <= i && i < count) {
      // constants
      const double er2 = 65536.0;
      const int     elements = 5;
      const int     channels = 4;
      // unpack state
      int ii = i * elements;
      double index = input[ii + 0];
      int i0 = index;
      double d0x = (deltaX + ((i0 % width) - (width /2))) * pixelSpacing;
      double d0y = (deltaY + ((i0 / width) - (height/2))) * pixelSpacing;
      double dx    = input[ii + 1];
```

```
    double dy    = input[ii + 2];
    double lx    = input[ii + 3];
    double ly    = input[ii + 4];
    bool escaped = false;
    int n = start;
    double yx = 0.0;
    double yy = 0.0;
    double y2 = 0.0;
    double dydcx = 0.0;
    double dydcy = 0.0;
    for (int m = 0; m < iters; ++m) {
      int p = n % period;
      double zx = reference[4*p+0];
      double zy = reference[4*p+1];
      double ax = reference[4*p+2];
      double ay = reference[4*p+3];
      // y <- z + d
      yx = zx + dx;
      yy = zy + dy;
      y2 = yx * yx + yy * yy;
      ++n;
      if (! (y2 <= er2)) {
        escaped = true;
        dydcx = (ax + lx) * pixelSpacing;
        dydcy = (ay + ly) * pixelSpacing;
        break;
      }
      // d <- 2 z d + d d + d0
      double tx = 2.0 * (zx * dx - zy * dy) + (dx * dx - dy * dy) + d0x;
      double ty = 2.0 * (zx * dy + zy * dx) + (2.0 * dx * dy) + d0y;
      // l <- 2 (a d + l d + l z)
      double sx = 2.0 * ((ax * dx - ay * dy) + (lx * dx - ly * dy) + (lx * zx - ↙
          ↳ ly * zy));
      double sy = 2.0 * ((ax * dy + ay * dx) + (lx * dy + ly * dx) + (lx * zy + ↙
          ↳ ly * zx));
      dx = tx;
      dy = ty;
      lx = sx;
      ly = sy;
    }
    // pack state
    if (escaped) {
      keep[i] = 0;
      keep[count] = 1;
      int jj = index * channels;
      double ly2 = log(y2);
      double ler2 = log(er2);
      double dydc2 = dydcx * dydcx + dydcy * dydcy;
      image[jj + 0] = n - log2(ly2 / ler2);
      image[jj + 1] = atan2(yy, yx);
      image[jj + 2] = sqrt(y2) * 0.5 * ly2 / sqrt(dydc2);
      image[jj + 3] = 0.0;
    } else {
      keep[i] = 1;
      output[ii + 0] = index;
      output[ii + 1] = dx;
      output[ii + 2] = dy;
```

```
      output[ii + 3] = lx;
      output[ii + 4] = ly;
    }
  }
}

__kernel void clear
  ( __global float *image
  ,          const int count
  ) {
  int i = get_global_id(0);
  if (0 <= i && i < count) {
    int jj = 4 * i;
    image[jj + 0] = -1.0;
    image[jj + 1] =  0.0;
    image[jj + 2] = -1.0;
    image[jj + 3] = -1.0;
  }
}

// linear interpolation by Robert Munafo mrob.com
float lin_interp
  ( float x
  , float domain_low
  , float domain_hi
  , float range_low
  , float range_hi
  ) {
  if ((x >= domain_low) && (x <= domain_hi)) {
    x = (x - domain_low) / (domain_hi - domain_low);
    x = range_low + x * (range_hi - range_low);
  }
  return x;
}

// perceptually balanced hue adjustment by Robert Munafo mrob.com
float pvp_adjust_3(float x) {
  x = lin_interp(x, 0.00, 0.125, -0.050, 0.090);
  x = lin_interp(x, 0.125, 0.25,  0.090, 0.167);
  x = lin_interp(x, 0.25, 0.375,  0.167, 0.253);
  x = lin_interp(x, 0.375, 0.50,  0.253, 0.383);
  x = lin_interp(x, 0.50, 0.625,  0.383, 0.500);
  x = lin_interp(x, 0.625, 0.75,  0.500, 0.667);
  x = lin_interp(x, 0.75, 0.875,  0.667, 0.800);
  x = lin_interp(x, 0.875, 1.00,  0.800, 0.950);
  return(x);
}

void hsv2rgb
  ( float h
  , float s
  , float v
  , float *rp
  , float *gp
  , float *bp
  ) {
  float i, f, p, q, t, r, g, b;
```

```
  if (s == 0.0) {
    r = v;
    g = v;
    b = v;
  } else {
    h = pvp_adjust_3(h);
    h = h - floor(h);
    h = h * 6.0;
    i = floor(h);
    f = h - i;
    p = v*(1.0 - s);
    q = v*(1.0 - (s*f));
    t = v*(1.0 - (s*(1.0 - f)));
    if (i < 1.0) { r = v; g = t; b = p; } else
    if (i < 2.0) { r = q; g = v; b = p; } else
    if (i < 3.0) { r = p; g = v; b = t; } else
    if (i < 4.0) { r = p; g = q; b = v; } else
    if (i < 5.0) { r = t; g = p; b = v; } else
                 { r = v; g = p; b = q; }
  }
  *rp = r;
  *gp = g;
  *bp = b;
}

__kernel void colour
  ( __global const float *image
  , __global unsigned char *rgb
  ,          const int count
  ) {
  int i = get_global_id(0);
  if (0 <= i && i < count) {
    int ii = 4 * i;
    int jj = 3 * i;
    float dwell = image[ii + 0];
    float distance = image[ii + 2];
    if (distance > 0.0) {
      float v = tanh(pow(distance/4.0f, 0.5f));
      float s = 0.5;
      float h = pow(dwell, 0.5f);
      h -= floor(h);
      float r, g, b;
      hsv2rgb(h, s, v, &r, &g, &b);
      rgb[jj + 0] = min(max(255.0 * r, 0.0), 255.0);
      rgb[jj + 1] = min(max(255.0 * g, 0.0), 255.0);
      rgb[jj + 2] = min(max(255.0 * b, 0.0), 255.0);
    } else {
      rgb[jj + 0] = 255;
      rgb[jj + 1] = 255;
      rgb[jj + 2] = 255;
    }
  }
}
```

31 mandelbrot-delta-cl/Makefile

```
mandelbrot-delta-cl: mandelbrot-delta-cl.cc cl-extra.cc
```

```
g++ -std=c++11 -Wall -pedantic -Wextra -O3 -o mandelbrot-delta-cl ↙
    ↳ mandelbrot-delta-cl.cc -lm -lmpfr -lOpenCL -lGL -lglfw -lGLEW
```

32 mandelbrot-delta-cl/mandelbrot-delta-cl.cc

```
// mandelbrot-delta-cl -- Mandelbrot set perturbation renderer using OpenCL
// Copyright: (C) 2013 Claude Heiland-Allen <claude@mathr.co.uk>
// License: http://www.gnu.org/licenses/gpl.html GPLv3+

// ==== [[[ config [[[ ====

// #define USE_DOUBLE to use double precision (highly recommended)
// #undef  USE_DOUBLE to use single precision (only for testing)
#define USE_DOUBLE

// #define USE_CLGL to use OpenCL -> OpenGL interoperation
// #undef  USE_CLGL to use copying fallback (eg: for CPU implementation)
#undef  USE_CLGL

// ==== ]]] config ]]] ====


#include <complex>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <boost/multiprecision/mpfr.hpp>
#include <GL/glew.h>
#include <GLFW/glfw3.h>
#include <GL/glx.h>
#include <CL/cl.h>
#include <CL/cl_gl.h>

#include "cl-extra.cc"

using namespace boost::multiprecision;
typedef mpfr_float R;
typedef std::complex<R> C;
#ifdef USE_DOUBLE
typedef double F;
#else
typedef float F;
#endif
typedef std::complex<F> CF;

// the OpenCL source

#ifdef USE_DOUBLE
#ifdef USE_CLGL
#define CL(cl) "#pragma OPENCL EXTENSION cl_khr_gl_sharing : enable\n#pragma ↙
    ↳ OPENCL EXTENSION cl_khr_fp64: enable\n" cl
#else
#define CL(cl) "#pragma OPENCL EXTENSION cl_khr_fp64: enable\n" cl
#endif
#else
#ifdef USE_CLGL
```

```
#define CL(cl) "#pragma OPENCL EXTENSION cl_khr_gl_sharing : enable\n" cl
#else
#define CL(cl) #cl
#endif
#endif

const char *src = CL("#include \"delta.cl\"");

int findPeriod(C c0, R r, int maxIters) {
  C c[4], z[4];
  c[0] = c0 + C(-r, -r);
  c[1] = c0 + C( r, -r);
  c[2] = c0 + C( r,  r);
  c[3] = c0 + C(-r,  r);
  z[0] = z[1] = z[2] = z[3] = C(0.0, 0.0);
  for (int p = 1; p < maxIters; ++p) {
    for (int i = 0; i < 4; ++i) {
      z[i] = z[i] * z[i] + c[i];
    }
    int preal = 0;
    for (int i = 0; i < 4; ++i) {
      int j = (i + 1) % 4;
      if (!(z[i].imag() < 0 && z[j].imag() < 0) && !(z[i].imag() > 0 && z[j].imag() > 0)) {
        C d = z[j] - z[i];
        if ((d.imag() * z[i].real() - d.real() * z[i].imag()) * d.imag() > 0) {
          ++preal;
        }
      }
    }
    if (preal & 1) {
      return p;
    }
  }
  return 0;
}

int muatom(int period, mpfr_t x, mpfr_t y, mpfr_t z) {
  const mpfr_prec_t accuracy = 12;
  mpfr_prec_t bits = std::max(mpfr_get_prec(x), mpfr_get_prec(y));
#define VARS nx, ny, bx, by, wx, wy, zz, Ax, Ay, Bx, By, Cx, Cy, Dx, Dy, Ex, Ey, t1, t2, t3, t4, t5, t6, t7, t8, t9, t0, t01, t02, t03, t04
  mpfr_t VARS;
  mpfr_inits2(bits, VARS, (mpfr_ptr) 0);
  mpfr_set(nx, x, MPFR_RNDN);
  mpfr_set(ny, y, MPFR_RNDN);
  mpfr_set_prec(zz, mpfr_get_prec(z));
  int n_ok, w_ok, b_ok;
  int r = 0;
  //progressReport('\n');
  do { //progressReport('X');
    mpfr_set_prec(t0,  bits - accuracy);
    mpfr_set_prec(t01, bits - accuracy);
    mpfr_set_prec(t02, bits - accuracy);
    mpfr_set_prec(t03, bits - accuracy);
    mpfr_set_prec(t04, bits - accuracy);
    int limit = 40;
```

```
    do { //progressReport('n');
      // refine nucleus
      mpfr_set_ui(Ax, 0, MPFR_RNDN);
      mpfr_set_ui(Ay, 0, MPFR_RNDN);
      mpfr_set_ui(Dx, 0, MPFR_RNDN);
      mpfr_set_ui(Dy, 0, MPFR_RNDN);
      for (int i = 0; i < period; ++i) {
        // D <- 2AD + 1   : Dx <- 2 (Ax Dx - Ay Dy) + 1 ; Dy = 2 (Ax Dy + Ay Dx)
        mpfr_mul(t1, Ax, Dx, MPFR_RNDN);
        mpfr_mul(t2, Ay, Dy, MPFR_RNDN);
        mpfr_mul(t3, Ax, Dy, MPFR_RNDN);
        mpfr_mul(t4, Ay, Dx, MPFR_RNDN);
        mpfr_sub(Dx, t1, t2, MPFR_RNDN);
        mpfr_mul_2ui(Dx, Dx, 1, MPFR_RNDN);
        mpfr_add_ui(Dx, Dx, 1, MPFR_RNDN);
        mpfr_add(Dy, t3, t4, MPFR_RNDN);
        mpfr_mul_2ui(Dy, Dy, 1, MPFR_RNDN);
        // A <- A^2 + n   : Ax <- Ax^2 - Ay^2 + nx ; Ay <- 2 Ax Ay + ny
        mpfr_sqr(t1, Ax, MPFR_RNDN);
        mpfr_sqr(t2, Ay, MPFR_RNDN);
        mpfr_mul(t3, Ax, Ay, MPFR_RNDN);
        mpfr_sub(Ax, t1, t2, MPFR_RNDN);
        mpfr_add(Ax, Ax, nx, MPFR_RNDN);
        mpfr_mul_2ui(t3, t3, 1, MPFR_RNDN);
        mpfr_add(Ay, t3, ny,MPFR_RNDN);
      }
      // n <- n - A / D  = (nx + ny i) - ((Ax Dx + Ay Dy) + (Ay Dx - Ax Dy)i) / ↙
          ↳ (Dx^2 + Dy^2)
      mpfr_sqr(t1, Dx, MPFR_RNDN);
      mpfr_sqr(t2, Dy, MPFR_RNDN);
      mpfr_add(t1, t1, t2, MPFR_RNDN);

      mpfr_mul(t2, Ax, Dx, MPFR_RNDN);
      mpfr_mul(t3, Ay, Dy, MPFR_RNDN);
      mpfr_add(t2, t2, t3, MPFR_RNDN);
      mpfr_div(t2, t2, t1, MPFR_RNDN);
      mpfr_sub(t2, nx, t2, MPFR_RNDN);

      mpfr_mul(t3, Ay, Dx, MPFR_RNDN);
      mpfr_mul(t4, Ax, Dy, MPFR_RNDN);
      mpfr_sub(t3, t3, t4, MPFR_RNDN);
      mpfr_div(t3, t3, t1, MPFR_RNDN);
      mpfr_sub(t3, ny, t3, MPFR_RNDN);

      mpfr_set(t01, t2, MPFR_RNDN);
      mpfr_set(t02, t3, MPFR_RNDN);
      mpfr_set(t03, nx, MPFR_RNDN);
      mpfr_set(t04, ny, MPFR_RNDN);
      mpfr_sub(t01, t01, t03, MPFR_RNDN);
      mpfr_sub(t02, t02, t04, MPFR_RNDN);
      n_ok = mpfr_zero_p(t01) && mpfr_zero_p(t02);

      mpfr_set(nx, t2, MPFR_RNDN);
      mpfr_set(ny, t3, MPFR_RNDN);
    } while (! n_ok && --limit);
    if (! limit) goto cleanup;
    mpfr_set(wx, nx, MPFR_RNDN);
```

```
      mpfr_set(wy, ny, MPFR_RNDN);
      mpfr_set(bx, nx, MPFR_RNDN);
      mpfr_set(by, ny, MPFR_RNDN);
      limit = 40;
      do { //progressReport('b');
        // refine bond
        mpfr_set(Ax, wx, MPFR_RNDN);
        mpfr_set(Ay, wy, MPFR_RNDN);
        mpfr_set_ui(Bx, 1, MPFR_RNDN);
        mpfr_set_ui(By, 0, MPFR_RNDN);
        mpfr_set_ui(Cx, 0, MPFR_RNDN);
        mpfr_set_ui(Cy, 0, MPFR_RNDN);
        mpfr_set_ui(Dx, 0, MPFR_RNDN);
        mpfr_set_ui(Dy, 0, MPFR_RNDN);
        mpfr_set_ui(Ex, 0, MPFR_RNDN);
        mpfr_set_ui(Ey, 0, MPFR_RNDN);
        for (int i = 0; i < period; ++i) {
          // E <- 2 ( A E + B D )
          mpfr_mul(t1, Ax, Ex, MPFR_RNDN);
          mpfr_mul(t2, Ay, Ey, MPFR_RNDN);
          mpfr_sub(t1, t1, t2, MPFR_RNDN);
          mpfr_mul(t2, Ax, Ey, MPFR_RNDN);
          mpfr_mul(t3, Ay, Ex, MPFR_RNDN);
          mpfr_add(t2, t2, t3, MPFR_RNDN);
          mpfr_mul(t3, Bx, Dx, MPFR_RNDN);
          mpfr_mul(t4, By, Dy, MPFR_RNDN);
          mpfr_sub(t3, t3, t4, MPFR_RNDN);
          mpfr_add(Ex, t1, t3, MPFR_RNDN);
          mpfr_mul(t3, Bx, Dy, MPFR_RNDN);
          mpfr_mul(t4, By, Dx, MPFR_RNDN);
          mpfr_add(t3, t3, t4, MPFR_RNDN);
          mpfr_add(Ey, t2, t3, MPFR_RNDN);
          mpfr_mul_2ui(Ex, Ex, 1, MPFR_RNDN);
          mpfr_mul_2ui(Ey, Ey, 1, MPFR_RNDN);
          // D <- 2 A D + 1
          mpfr_mul(t1, Ax, Dx, MPFR_RNDN);
          mpfr_mul(t2, Ay, Dy, MPFR_RNDN);
          mpfr_mul(t3, Ax, Dy, MPFR_RNDN);
          mpfr_mul(t4, Ay, Dx, MPFR_RNDN);
          mpfr_sub(Dx, t1, t2, MPFR_RNDN);
          mpfr_mul_2ui(Dx, Dx, 1, MPFR_RNDN);
          mpfr_add_ui(Dx, Dx, 1, MPFR_RNDN);
          mpfr_add(Dy, t3, t4, MPFR_RNDN);
          mpfr_mul_2ui(Dy, Dy, 1, MPFR_RNDN);
          // C <- 2 ( B^2 + A C )
          mpfr_sqr(t1, Bx, MPFR_RNDN);
          mpfr_sqr(t2, By, MPFR_RNDN);
          mpfr_sub(t1, t1, t2, MPFR_RNDN);
          mpfr_mul(t2, Bx, By, MPFR_RNDN);
          mpfr_mul_2ui(t2, t2, 1, MPFR_RNDN);
          mpfr_mul(t3, Ax, Cx, MPFR_RNDN);
          mpfr_mul(t4, Ay, Cy, MPFR_RNDN);
          mpfr_sub(t3, t3, t4, MPFR_RNDN);
          mpfr_add(t1, t1, t3, MPFR_RNDN);
          mpfr_mul(t3, Ax, Cy, MPFR_RNDN);
          mpfr_mul(t4, Ay, Cx, MPFR_RNDN);
          mpfr_add(t3, t3, t4, MPFR_RNDN);
```

```
220             mpfr_add(t2, t2, t3, MPFR_RNDN);
                mpfr_mul_2ui(Cx, t1, 1, MPFR_RNDN);
                mpfr_mul_2ui(Cy, t2, 1, MPFR_RNDN);
                // B <- 2 A B
                mpfr_mul(t1, Ax, Bx, MPFR_RNDN);
225             mpfr_mul(t2, Ay, By, MPFR_RNDN);
                mpfr_mul(t3, Ax, By, MPFR_RNDN);
                mpfr_mul(t4, Ay, Bx, MPFR_RNDN);
                mpfr_sub(Bx, t1, t2, MPFR_RNDN);
                mpfr_mul_2ui(Bx, Bx, 1, MPFR_RNDN);
230             mpfr_add(By, t3, t4, MPFR_RNDN);
                mpfr_mul_2ui(By, By, 1, MPFR_RNDN);
                // A <- A^2 + b
                mpfr_sqr(t1, Ax, MPFR_RNDN);
                mpfr_sqr(t2, Ay, MPFR_RNDN);
235             mpfr_mul(t3, Ax, Ay, MPFR_RNDN);
                mpfr_sub(Ax, t1, t2, MPFR_RNDN);
                mpfr_add(Ax, Ax, bx, MPFR_RNDN);
                mpfr_mul_2ui(t3, t3, 1, MPFR_RNDN);
                mpfr_add(Ay, t3, by,MPFR_RNDN);
240         }
            // (w) <- (w) - (B-1  D)^-1 (A - w)
            // (b) <- (b)    ( C   E)     (B + 1)
            //
            // det = (B-1)E - CD
245         // inv = (E    -D)
            //       (-C (B-1) / det
            //
            // w - (E(A-w) - D(B+1))/det
            // b - (-C(A-w) + (B-1)(B+1))/det    ; B^2 - 1
250         //
            // A -= w
            mpfr_sub(Ax, Ax, wx, MPFR_RNDN);
            mpfr_sub(Ay, Ay, wy, MPFR_RNDN);
            // (t1,t2) = B^2 - 1
255         mpfr_mul(t1, Bx, Bx, MPFR_RNDN);
            mpfr_mul(t2, By, By, MPFR_RNDN);
            mpfr_sub(t1, t1, t2, MPFR_RNDN);
            mpfr_sub_ui(t1, t1, 1, MPFR_RNDN);
            mpfr_mul(t2, Bx, By, MPFR_RNDN);
260         mpfr_mul_2ui(t2, t2, 1, MPFR_RNDN);
            // (t1,t2) -= C(A-w)
            mpfr_mul(t3, Cx, Ax, MPFR_RNDN);
            mpfr_mul(t4, Cy, Ay, MPFR_RNDN);
            mpfr_sub(t3, t3, t4, MPFR_RNDN);
265         mpfr_sub(t1, t1, t3, MPFR_RNDN);
            mpfr_mul(t3, Cx, Ay, MPFR_RNDN);
            mpfr_mul(t4, Cy, Ax, MPFR_RNDN);
            mpfr_add(t3, t3, t4, MPFR_RNDN);
            mpfr_sub(t2, t2, t3, MPFR_RNDN);
270         // (t3, t4) = (B-1)E - CD
            mpfr_sub_ui(t3, Bx, 1, MPFR_RNDN);
            mpfr_mul(t4, t3, Ex, MPFR_RNDN);
            mpfr_mul(t5, By, Ey, MPFR_RNDN);
            mpfr_sub(t4, t4, t5, MPFR_RNDN);
275         mpfr_mul(t5, t3, Ey, MPFR_RNDN);
            mpfr_mul(t6, By, Ex, MPFR_RNDN);
```

```
        mpfr_sub(t5, t5, t6, MPFR_RNDN);
        mpfr_mul(t6, Cx, Dx, MPFR_RNDN);
        mpfr_mul(t7, Cy, Dy, MPFR_RNDN);
        mpfr_sub(t6, t6, t7, MPFR_RNDN);
        mpfr_sub(t3, t4, t6, MPFR_RNDN);
        mpfr_mul(t6, Cx, Dy, MPFR_RNDN);
        mpfr_mul(t7, Cy, Dx, MPFR_RNDN);
        mpfr_add(t6, t6, t7, MPFR_RNDN);
        mpfr_sub(t4, t5, t6, MPFR_RNDN);
        // (t3, t4) = 1/(t3, t4)   ; z^-1 = z* / (z z*)
        mpfr_sqr(t5, t3, MPFR_RNDN);
        mpfr_sqr(t6, t4, MPFR_RNDN);
        mpfr_add(t5, t5, t6, MPFR_RNDN);
        mpfr_div(t3, t3, t5, MPFR_RNDN);
        mpfr_div(t4, t4, t5, MPFR_RNDN);
        mpfr_neg(t4, t4, MPFR_RNDN);
        // (t1, t2) *= (t3, t4)
        mpfr_mul(t5, t1, t3, MPFR_RNDN);
        mpfr_mul(t6, t2, t4, MPFR_RNDN);
        mpfr_mul(t7, t1, t4, MPFR_RNDN);
        mpfr_mul(t8, t2, t3, MPFR_RNDN);
        mpfr_sub(t1, t5, t6, MPFR_RNDN);
        mpfr_add(t2, t7, t8, MPFR_RNDN);

        // (t1, t2) = b - (t1, t2)
        mpfr_sub(t1, bx, t1, MPFR_RNDN);
        mpfr_sub(t2, by, t2, MPFR_RNDN);

        mpfr_set(t01, t1, MPFR_RNDN);
        mpfr_set(t02, t2, MPFR_RNDN);
        mpfr_set(t03, bx, MPFR_RNDN);
        mpfr_set(t04, by, MPFR_RNDN);
        mpfr_sub(t01, t01, t03, MPFR_RNDN);
        mpfr_sub(t02, t02, t04, MPFR_RNDN);
        b_ok = mpfr_zero_p(t01) && mpfr_zero_p(t02);

        // (t5, t6) = E (A-w)
        mpfr_mul(t5, Ex, Ax, MPFR_RNDN);
        mpfr_mul(t6, Ey, Ay, MPFR_RNDN);
        mpfr_sub(t5, t5, t6, MPFR_RNDN);
        mpfr_mul(t6, Ex, Ay, MPFR_RNDN);
        mpfr_mul(t7, Ey, Ax, MPFR_RNDN);
        mpfr_add(t6, t6, t7, MPFR_RNDN);
        // B += 1
        mpfr_add_ui(Bx, Bx, 1, MPFR_RNDN);
        // (t7, t8) = D (B+1)
        mpfr_mul(t7, Dx, Bx, MPFR_RNDN);
        mpfr_mul(t8, Dy, By, MPFR_RNDN);
        mpfr_sub(t7, t7, t8, MPFR_RNDN);
        mpfr_mul(t8, Dx, By, MPFR_RNDN);
        mpfr_mul(t9, Dy, Bx, MPFR_RNDN);
        mpfr_add(t8, t8, t9, MPFR_RNDN);
        // (t5, t6) -= (t7, t8)
        mpfr_sub(t5, t5, t7, MPFR_RNDN);
        mpfr_sub(t6, t6, t8, MPFR_RNDN);
        // (t7, t8) = (t5, t6) * (t3, t4)
        mpfr_mul(t7, t3, t5, MPFR_RNDN);
```

```
        mpfr_mul(t8, t4, t6, MPFR_RNDN);
        mpfr_sub(t7, t7, t8, MPFR_RNDN);
        mpfr_mul(t8, t3, t6, MPFR_RNDN);
        mpfr_mul(t9, t4, t5, MPFR_RNDN);
        mpfr_add(t8, t8, t9, MPFR_RNDN);

        // (t3, t4) = w - (t7, t8)
        mpfr_sub(t3, wx, t7, MPFR_RNDN);
        mpfr_sub(t4, wy, t8, MPFR_RNDN);

        mpfr_set(t01, t3, MPFR_RNDN);
        mpfr_set(t02, t4, MPFR_RNDN);
        mpfr_set(t03, wx, MPFR_RNDN);
        mpfr_set(t04, wy, MPFR_RNDN);
        mpfr_sub(t01, t01, t03, MPFR_RNDN);
        mpfr_sub(t02, t02, t04, MPFR_RNDN);
        w_ok = mpfr_zero_p(t01) && mpfr_zero_p(t02);

        mpfr_set(t01, t3, MPFR_RNDN);
        mpfr_set(t02, t4, MPFR_RNDN);
        mpfr_set(t03, wx, MPFR_RNDN);
        mpfr_set(t04, wy, MPFR_RNDN);
        mpfr_sub(t01, t01, t03, MPFR_RNDN);
        mpfr_sub(t02, t02, t04, MPFR_RNDN);
        w_ok = mpfr_zero_p(t01) && mpfr_zero_p(t02);

        mpfr_set(bx, t1, MPFR_RNDN);
        mpfr_set(by, t2, MPFR_RNDN);
        mpfr_set(wx, t3, MPFR_RNDN);
        mpfr_set(wy, t4, MPFR_RNDN);
      } while (!(w_ok && b_ok) && --limit);
      if (! limit) goto cleanup;
      // t1 = |n - b|
      mpfr_sub(t1, nx, bx, MPFR_RNDN);
      mpfr_sqr(t1, t1, MPFR_RNDN);
      mpfr_sub(t2, ny, by, MPFR_RNDN);
      mpfr_sqr(t2, t2, MPFR_RNDN);
      mpfr_add(t1, t1, t2, MPFR_RNDN);
      mpfr_sqrt(t1, t1, MPFR_RNDN);
      bits = 2 * bits;
      mpfr_prec_round(nx, bits, MPFR_RNDN);
      mpfr_prec_round(ny, bits, MPFR_RNDN);
      mpfr_prec_round(wx, bits, MPFR_RNDN);
      mpfr_prec_round(wy, bits, MPFR_RNDN);
      mpfr_prec_round(bx, bits, MPFR_RNDN);
      mpfr_prec_round(by, bits, MPFR_RNDN);
      mpfr_set(zz, t1, MPFR_RNDN);
      mpfr_set_prec(Ax, bits);
      mpfr_set_prec(Ay, bits);
      mpfr_set_prec(Bx, bits);
      mpfr_set_prec(By, bits);
      mpfr_set_prec(Cx, bits);
      mpfr_set_prec(Cy, bits);
      mpfr_set_prec(Dx, bits);
      mpfr_set_prec(Dy, bits);
      mpfr_set_prec(Ex, bits);
      mpfr_set_prec(Ey, bits);
```

```
      mpfr_set_prec(t1, bits);
      mpfr_set_prec(t2, bits);
      mpfr_set_prec(t3, bits);
      mpfr_set_prec(t4, bits);
      mpfr_set_prec(t5, bits);
      mpfr_set_prec(t6, bits);
      mpfr_set_prec(t7, bits);
      mpfr_set_prec(t8, bits);
      mpfr_set_prec(t9, bits);
    } while (mpfr_zero_p(zz) || bits + mpfr_get_exp(zz) < mpfr_get_prec(zz) + accuracy);

    // copy to output
    r = 1;
    mpfr_set_prec(x, bits);
    mpfr_set_prec(y, bits);
    mpfr_set(x, nx, MPFR_RNDN);
    mpfr_set(y, ny, MPFR_RNDN);
    mpfr_set(z, zz, MPFR_RNDN);
  cleanup:
    mpfr_clears(VARS, (mpfr_ptr) 0);
  #undef VARS
    return r;
  }

  struct atom {
    C nucleus;
    R radius;
    int period;
  };

  struct atom *findAtom(C c, R r, int period) {
    struct atom *a = 0;
    mpfr_t x;
    mpfr_t y;
    mpfr_t z;
    mpfr_inits2(mpfr_get_prec(c.real().backend().data()), x, y, (mpfr_ptr) 0);
    mpfr_inits2(mpfr_get_prec(r.backend().data()), z, (mpfr_ptr) 0);
    mpfr_set(x, c.real().backend().data(), MPFR_RNDN);
    mpfr_set(y, c.imag().backend().data(), MPFR_RNDN);
    mpfr_set(z, r.backend().data(), MPFR_RNDN);
    int ok = muatom(period, x, y, z);
    if (ok) {
      int prec = int(ceil(log10(2.0) * mpfr_get_prec(x)));
      R::default_precision(prec);
      a = new atom;
      a->nucleus.real().precision(prec);
      a->nucleus.imag().precision(prec);
      a->nucleus = C(R(x), R(y));
      a->radius  = R(z);
      a->period  = period;
    }
    mpfr_clears(x, y, z, (mpfr_ptr) 0);
    return a;
  }

  // the main program
```

```
int main(int argc, char **argv) {

  // read command line arguments
  if (argc != 9) {
    fprintf(stderr, "usage: %s  platform  width height  re im size  polyiters ↙
        ↳ maxiters >out.ppm\n", argv[0]);
    return 1;
  }
  cl_uint PLATFORM = atoi(argv[1]);
  cl_int w(atoi(argv[2]));
  cl_int h(atoi(argv[3]));
  R::default_precision(20);
  F csize(atof(argv[6]));
  R::default_precision(int(20 - log10(csize)));
  F pixelSpacing = csize / (h/2);
  R cx(argv[4]);
  R cy(argv[5]);
  C c(cx, cy);
  cl_int precount(atoi(argv[7]));
  cl_int maxIters(atoi(argv[8]));

  // compute period
  cl_int period = findPeriod(c, csize, maxIters);
  if (! period) {
    fprintf(stderr, "failed to find period\n");
    return 1;
  }
  fprintf(stderr, "period = %d\n", period);

  // compute atom
  struct atom *ref = 0;
  if (! (ref = findAtom(c, csize, period))) {
    fprintf(stderr, "failed to find reference\n");
    return 1;
  }
  mpfr_fprintf(stderr
    , "real = %Re\nimag = %Re\nsize = %Re\n"
    , ref->nucleus.real().backend().data()
    , ref->nucleus.imag().backend().data()
    , ref->radius.backend().data()
    );

  R::default_precision(ref->nucleus.real().precision());
  C refc = ref->nucleus;
  cl_int iters = 0;
  cl_int stepIters = 1024;

  // compute one period at high precision
  int per_bytes = 4 * period * sizeof(F);
  F *per_zs = (F *) calloc(1, per_bytes);
  {
    C c0(refc);
    C z(c0);
    C a(1.0, 0.0);
    for (int i = 0; i < period; ++i) {
      if (i == period - 1) {
```

```
        z = C(0.0, 0.0);
      }
      per_zs[4 * i + 0] = F(z.real());
      per_zs[4 * i + 1] = F(z.imag());
      per_zs[4 * i + 2] = F(a.real());
      per_zs[4 * i + 3] = F(a.imag());
      a = C(2.0, 0.0) * z * a + C(1.0, 0.0);
      z = z * z + c0;
    }
  }

  // storage for order-12 polynomial approximation
  int pre_bytes = 12 * 4 * sizeof(F);
  F *pre_zs = (F *) calloc(1, pre_bytes);

  // initialize OpenGL
  if (! glfwInit()) {
    fprintf(stderr, "glfwInit() failed\n");
    return 1;
  };
  glfwWindowHint(GLFW_RESIZABLE, GL_FALSE);
  GLFWwindow *window = glfwCreateWindow(w, h, "MandelbrotDeltaCL", 0, 0);
  if (! window) {
    fprintf(stderr, "glfwOpenWindow() failed\n");
    return 1;
  }
  glfwMakeContextCurrent(window);
  glewInit();

  // enumerate OpenCL platforms
  cl_platform_id platform_id[64];
  cl_uint platform_ids;
  E(clGetPlatformIDs(64, &platform_id[0], &platform_ids));
  if (! (PLATFORM < platform_ids)) {
    fprintf(stderr, "invalid platform: %d (range is 0 to %d)\n", PLATFORM, ↙
        ↳ platform_ids - 1);
    return 1;
  }
  for (cl_uint i = 0; i < platform_ids; ++i) {
    fprintf(stderr, "platform %d%s\n", i, i == PLATFORM ? " *** SELECTED ***" : ↙
        ↳ "");
    char buf[1024];
    cl_platform_info info[5] =
      { CL_PLATFORM_PROFILE
      , CL_PLATFORM_VERSION
      , CL_PLATFORM_NAME
      , CL_PLATFORM_VENDOR
      , CL_PLATFORM_EXTENSIONS
      };
    for (int j = 0; j < 5; ++j) {
      buf[0] = 0;
      E(clGetPlatformInfo(platform_id[i], info[j], 1024, &buf[0], NULL));
      fprintf(stderr, "\t%s\n", buf);
    }
  }

  // create context
```

```
  cl_device_id device_id;
  E(clGetDeviceIDs(platform_id[PLATFORM], CL_DEVICE_TYPE_ALL, 1, &device_id, ↙
      ↳ NULL));
560   cl_context_properties properties[] =
    {
#ifdef USE_CLGL
      CL_GL_CONTEXT_KHR, (cl_context_properties) glXGetCurrentContext(),
      CL_GLX_DISPLAY_KHR, (cl_context_properties) glXGetCurrentDisplay(),
565 #endif
      CL_CONTEXT_PLATFORM, (cl_context_properties) platform_id[PLATFORM]
    , 0
    };
  cl_int err;
570   cl_context context = clCreateContext(properties, 1, &device_id, NULL, NULL, &↙
      ↳ err);
  if (! context) { E(err); }
  cl_command_queue commands = clCreateCommandQueue(context, device_id, 0, &err);
  if (! commands) { E(err); }

575   // compile program
  cl_program program = clCreateProgramWithSource(context, 1, &src, NULL, &err);
  if (! program) { E(err); }
  err = clBuildProgram(program, 1, &device_id, "-I. -cl-finite-math-only -cl-no-↙
      ↳ signed-zeros"
#ifdef USE_DOUBLE
580     ""
#else
    " -Ddouble=float"
#endif
    , NULL, NULL);
585   if (err != CL_SUCCESS) {
    char *buf = (char *) malloc(1000000);
    buf[0] = 0;
    E(clGetProgramBuildInfo(program, device_id, CL_PROGRAM_BUILD_LOG, 1000000, &↙
        ↳ buf[0], NULL));
    fprintf(stderr, "build failed:\n%s\n", buf);
590     free(buf);
    E(err);
  }

  // create kernels
595   cl_kernel prefixsum = clCreateKernel(program, "prefixsum", &err);
  if (! prefixsum) { E(err); }
  cl_kernel permute = clCreateKernel(program, "permute", &err);
  if (! permute) { E(err); }
  cl_kernel initialize = clCreateKernel(program, "initialize", &err);
600   if (! initialize) { E(err); }
  cl_kernel iterate = clCreateKernel(program, "iterate", &err);
  if (! iterate) { E(err); }
  cl_kernel clear = clCreateKernel(program, "clear", &err);
  if (! clear) { E(err); }
605   cl_kernel colour = clCreateKernel(program, "colour", &err);
  if (! colour) { E(err); }

  // create buffers
  cl_mem in_pre_zs = clCreateBuffer(context, CL_MEM_READ_ONLY, pre_bytes, NULL, ↙
      ↳ &err);
```

```
610     if (! in_pre_zs) { E(err); }

        cl_mem in_per_zs = clCreateBuffer(context, CL_MEM_READ_ONLY, per_bytes, NULL, ↙
            ↳ &err);
        if (! in_per_zs) { E(err); }
        E(clEnqueueWriteBuffer(commands, in_per_zs, CL_TRUE, 0, per_bytes, &per_zs[0],↙
            ↳  0, NULL, NULL));
615
        cl_int elements = 5;
        int ebytes = w * h * elements * sizeof(F);
        cl_mem state[2];
        state[0] = clCreateBuffer(context, CL_MEM_READ_WRITE, ebytes, NULL, &err);
620     if (! state[0]) { E(err); }
        state[1] = clCreateBuffer(context, CL_MEM_READ_WRITE, ebytes, NULL, &err);
        if (! state[1]) { E(err); }

        int kbytes = (w * h + 1) * sizeof(cl_int);
625     cl_mem keep = clCreateBuffer(context, CL_MEM_READ_WRITE, kbytes, NULL, &err);
        if (! keep) { E(err); }

        int sbytes = (w * h) * sizeof(cl_int);
        cl_mem sums[2];
630     sums[0] = clCreateBuffer(context, CL_MEM_READ_WRITE, sbytes, NULL, &err);
        if (! sums[0]) { E(err); }
        sums[1] = clCreateBuffer(context, CL_MEM_READ_WRITE, sbytes, NULL, &err);
        if (! sums[1]) { E(err); }

635     int channels = 4;
        int ibytes = w * h * channels * sizeof(float);
        cl_mem image = clCreateBuffer(context, CL_MEM_WRITE_ONLY, ibytes, NULL, &err);
        if (! image) { E(err); }

640     int gbytes = 3 * w * h;
        unsigned char *rgb = (unsigned char *) calloc(1, gbytes);
        GLuint texture;
        glGenTextures(1, &texture);
        glBindTexture(GL_TEXTURE_2D, texture);
645     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_BASE_LEVEL, 0);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAX_LEVEL, 0);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, w, h, 0, GL_RGB, GL_UNSIGNED_BYTE, NULL↙
            ↳ );
650     glEnable(GL_TEXTURE_2D);
    #ifdef USE_CLGL
        GLuint vbo;
        glGenBuffers(1, &vbo);
        glBindBuffer(GL_PIXEL_UNPACK_BUFFER, vbo);
655     glBufferData(GL_PIXEL_UNPACK_BUFFER, gbytes, 0, GL_DYNAMIC_DRAW);
        err = glGetError();
        glFinish();
        cl_mem out_rgb = clCreateFromGLBuffer(context, CL_MEM_READ_WRITE, vbo, &err);
        if (! out_rgb) { E(err); }
660     cl_event acquired;
        E(clEnqueueAcquireGLObjects(commands, 1, &out_rgb, 0, NULL, &acquired));
        clWaitForEvents(1, &acquired);
    #else
```

```
  cl_mem out_rgb = clCreateBuffer(context, CL_MEM_READ_WRITE, gbytes, NULL, &err↙
      ↳ );
  if (! out_rgb) { E(err); }
#endif

  C delta(c - refc);
  F deltaX(R(delta.real() / R(pixelSpacing)));
  F deltaY(R(delta.imag() / R(pixelSpacing)));
  R radius((abs(delta) + abs(C(1.0 * w, 1.0 * h))) * pixelSpacing);

  // print viewing parameters
  mpfr_fprintf(stderr, "# %dx%d\n# re = ", w, h);
  mpfr_out_str(stderr, 10, 0, c.real().backend().data(), MPFR_RNDN);
  mpfr_fprintf(stderr, "\n# im = ");
  mpfr_out_str(stderr, 10, 0, c.imag().backend().data(), MPFR_RNDN);
  mpfr_fprintf(stderr, "\n# @    %g    P%d p%d\n# rre ", csize, period, precount↙
      ↳ );
  mpfr_out_str(stderr, 10, 0, refc.real().backend().data(), MPFR_RNDN);
  mpfr_fprintf(stderr, "\n# rim ");
  mpfr_out_str(stderr, 10, 0, refc.imag().backend().data(), MPFR_RNDN);
  mpfr_fprintf(stderr, "\n");

  // compute high precision polynomial approximation and round to F53
  {
    C c0(refc);
    C z(c0);

    C a(1.0, 0.0);
    C b(0.0, 0.0);
    C c(0.0, 0.0);
    C d(0.0, 0.0);
    C e(0.0, 0.0);
    C f(0.0, 0.0);
    C g(0.0, 0.0);
    C h(0.0, 0.0);
    C i(0.0, 0.0);
    C j(0.0, 0.0);
    C k(0.0, 0.0);
    C l(0.0, 0.0);
    C aa(0.0, 0.0);
    C bb(0.0, 0.0);
    C cc(0.0, 0.0);
    C dd(0.0, 0.0);
    C ee(0.0, 0.0);
    C ff(0.0, 0.0);
    C gg(0.0, 0.0);
    C hh(0.0, 0.0);
    C ii(0.0, 0.0);
    C jj(0.0, 0.0);
    C kk(0.0, 0.0);
    C ll(0.0, 0.0);

    R r(csize);
    R rr(r);
    R two(2.0);
    C one(1.0, 0.0);
    for (int iii = 0; iii < precount; ++iii) {
```

```
      if (iii % period == period - 1) {
        z = C(0.0, 0.0);
      }
      ll = two * (ll*z+a*l+a*kk+aa*k+b*jj+bb*j+c*ii+cc*i+d*hh+dd*h+e*gg+ee*g+f*ff);
      kk = two * (kk*z+a*k+a*jj+aa*j+b*ii+bb*i+c*hh+cc*h+d*gg+dd*g+e*ff+ee*f);
      jj = two * (jj*z+a*j+a*ii+aa*i+b*hh+bb*h+c*gg+cc*g+d*ff+dd*f+e*ee);
      ii = two * (ii*z+a*i+a*hh+aa*h+b*gg+bb*g+c*ff+cc*f+d*ee+dd*e);
      hh = two * (hh*z+a*h+a*gg+aa*g+b*ff+bb*f+c*ee+cc*e+d*dd);
      gg = two * (gg*z+a*g+a*ff+aa*f+b*ee+bb*e+c*dd+cc*d);
      ff = two * (ff*z+a*f+a*ee+aa*e+b*dd+bb*d+c*cc);
      ee = two * (ee*z+a*e+a*dd+aa*d+b*cc+bb*c);
      dd = two * (dd*z+a*d+a*cc+aa*c+b*bb);
      cc = two * (cc*z+a*c+a*bb+aa*b);
      bb = two * (bb*z+a*b+a*aa);
      aa = two * (aa*z+a*a);
      l = two * (l*z+a*k+b*j+c*i+d*h+e*g) + f*f;
      k = two * (k*z+a*j+b*i+c*h+d*g+e*f);
      j = two * (j*z+a*i+b*h+c*g+d*f) + e*e;
      i = two * (i*z+a*h+b*g+c*f+d*e);
      h = two * (h*z+a*g+b*f+c*e) + d*d;
      g = two * (g * z + a * f + b * e + c * d);
      f = two * (f * z + a * e + b * d) + c * c;
      e = two * (e * z + a * d + b * c);
      d = two * (d * z + a * c) + b * b;
      c = two * (c * z + a * b);
      b = two * b * z + a * a;
      a = two * a * z + one;
      z = z * z + c0;
    }
    int idx = 0;
#define S(U,V) \
          pre_zs[idx++]=F(R(U.real()*rr)); \
          pre_zs[idx++]=F(R(U.imag()*rr)); \
          pre_zs[idx++]=F(R(V.real()*rr)); \
          pre_zs[idx++]=F(R(V.imag()*rr)); \
          rr *= r;
    S(a,aa)S(b,bb)S(c,cc)S(d,dd)S(e,ee)S(f,ff)S(g,gg)S(h,hh)S(i,ii)S(j,jj)S(k,kk)S(l,ll)
#undef S
  }
  E(clEnqueueWriteBuffer(commands, in_pre_zs, CL_TRUE, 0, pre_bytes, &pre_zs[0], 0, NULL, NULL));

  // initialize kernel arguments
  cl_int count = w * h;
  size_t local, global;

  E(clSetKernelArg(clear, 0, sizeof(cl_mem), &image));
  E(clSetKernelArg(clear, 1, sizeof(cl_int), &count));
  E(clGetKernelWorkGroupInfo(clear, device_id, CL_KERNEL_WORK_GROUP_SIZE, sizeof(local), &local, NULL));
  global = ((count + local - 1)/local) * local;
  cl_event cleared;
  E(clEnqueueNDRangeKernel(commands, clear, 1, NULL, &global, &local, 0, NULL, &cleared));
```

```
  int state_which = 0;
  E(clSetKernelArg(initialize, 0, sizeof(cl_mem), &state[state_which]));
  E(clSetKernelArg(initialize, 1, sizeof(cl_int), &w));
  E(clSetKernelArg(initialize, 2, sizeof(cl_int), &h));
  E(clSetKernelArg(initialize, 3, sizeof(cl_mem), &in_pre_zs));
  E(clSetKernelArg(initialize, 4, sizeof(F), &deltaX));
  E(clSetKernelArg(initialize, 5, sizeof(F), &deltaY));
  E(clGetKernelWorkGroupInfo(initialize, device_id, CL_KERNEL_WORK_GROUP_SIZE, ↙
      ↳ sizeof(local), &local, NULL));
  global = ((count + local - 1)/local) * local;
  cl_event initialized;
  E(clEnqueueNDRangeKernel(commands, initialize, 1, NULL, &global, &local, 1, &↙
      ↳ cleared, &initialized));

  E(clSetKernelArg(iterate, 0, sizeof(cl_mem), &in_per_zs));
  E(clSetKernelArg(iterate, 3, sizeof(cl_mem), &keep));
  E(clSetKernelArg(iterate, 4, sizeof(cl_mem), &image));
  E(clSetKernelArg(iterate, 5, sizeof(cl_int), &count));
  E(clSetKernelArg(iterate, 6, sizeof(cl_int), &w));
  E(clSetKernelArg(iterate, 7, sizeof(cl_int), &h));
  E(clSetKernelArg(iterate, 8, sizeof(F), &deltaX));
  E(clSetKernelArg(iterate, 9, sizeof(F), &deltaY));
  E(clSetKernelArg(iterate, 10, sizeof(F), &pixelSpacing));
  E(clSetKernelArg(iterate, 12, sizeof(cl_int), &stepIters));
  E(clSetKernelArg(iterate, 13, sizeof(cl_int), &period));

  E(clSetKernelArg(permute, 1, sizeof(cl_mem), &keep));
  E(clSetKernelArg(permute, 5, sizeof(cl_int), &elements));

  cl_event caniterate = initialized;
  iters = precount;

  // iterate until rate of escapes is low
  bool anyEscape = false;
  bool recentEscape = false;
  cl_int targetIters = iters + stepIters;
  cl_int oldCount = count;
  bool done = false;
  while (! done) {
    fprintf(stderr, "%16d%16d%16d\n", count, iters, state_which);

    E(clSetKernelArg(iterate, 1, sizeof(cl_mem), &state[state_which]));
    E(clSetKernelArg(iterate, 2, sizeof(cl_mem), &state[1 - state_which]));
    E(clSetKernelArg(permute, 0, sizeof(cl_mem), &state[1 - state_which]));
    E(clSetKernelArg(permute, 3, sizeof(cl_mem), &state[state_which]));

    // iterate
    cl_int zero = 0;
    E(clEnqueueWriteBuffer(commands, keep, CL_TRUE, count * sizeof(cl_int), ↙
        ↳ sizeof(cl_int), &zero, 0, NULL, NULL));
    E(clSetKernelArg(iterate, 5, sizeof(cl_int), &count));
    E(clSetKernelArg(iterate, 11, sizeof(cl_int), &iters));
    E(clGetKernelWorkGroupInfo(iterate, device_id, CL_KERNEL_WORK_GROUP_SIZE, ↙
        ↳ sizeof(local), &local, NULL));
    global = ((count + local - 1)/local) * local;
    cl_event iterated;
    E(clEnqueueNDRangeKernel(commands, iterate, 1, NULL, &global, &local, 1, &↙
```

```
    ↪ caniterate, &iterated));
  cl_int any_escaped = 0;
825 E(clEnqueueReadBuffer(commands, keep, CL_TRUE, count * sizeof(cl_int), ↙
    ↪ sizeof(cl_int), &any_escaped, 1, &iterated, NULL));
  iters += stepIters;

  // compact working set if necessary, otherwise ping pong state buffers
  if (any_escaped) {
830 // prefix sum
    E(clSetKernelArg(prefixsum, 0, sizeof(cl_mem), &keep));
    E(clSetKernelArg(prefixsum, 2, sizeof(cl_int), &count));
    E(clGetKernelWorkGroupInfo(prefixsum, device_id, CL_KERNEL_WORK_GROUP_SIZE↙
      ↪ , sizeof(local), &local, NULL));
    global = ((count + local - 1)/local) * local;
835 int pswhich = 0;
    cl_event lastsummed = iterated;
    for (cl_int passbit = 1; passbit <= count; passbit <<= 1) {
      E(clSetKernelArg(prefixsum, 1, sizeof(cl_mem), &sums[pswhich]));
      E(clSetKernelArg(prefixsum, 3, sizeof(cl_int), &passbit));
840 cl_event summed;
      E(clEnqueueNDRangeKernel(commands, prefixsum, 1, NULL, &global, &local, ↙
        ↪ 1, &lastsummed, &summed));
      lastsummed = summed;
      E(clSetKernelArg(prefixsum, 0, sizeof(cl_mem), &sums[pswhich]));
      E(clSetKernelArg(permute, 2, sizeof(cl_mem), &sums[pswhich]));
845 pswhich = 1 - pswhich;
    }
    // permute
    E(clSetKernelArg(permute, 4, sizeof(cl_int), &count));
    E(clGetKernelWorkGroupInfo(permute, device_id, CL_KERNEL_WORK_GROUP_SIZE, ↙
      ↪ sizeof(local), &local, NULL));
850 global = ((count + local - 1)/local) * local;
    cl_event permuted;
    E(clEnqueueNDRangeKernel(commands, permute, 1, NULL, &global, &local, 1, &↙
      ↪ lastsummed, &permuted));
    // read back count
    E(clEnqueueReadBuffer(commands, sums[1 - pswhich], CL_TRUE, (count - 1) * ↙
      ↪ sizeof(cl_int), sizeof(cl_int), &count, 1, &lastsummed, NULL));
855 caniterate = permuted;
  } else {
    state_which = 1 - state_which;
    caniterate = iterated;
  }
860
  // check if done
  if (count < oldCount) {
    anyEscape = true;
    recentEscape = true;
865 }
  done = count == 0;
  if (targetIters <= iters) {
    done = count == 0 || (anyEscape ? !recentEscape : false);
    oldCount = count;
870 recentEscape = false;
    targetIters <<= 1;
  }
  if (maxIters <= iters) {
```

```
        done = true;
875     }
      }

      // colourize
      count = w * h;
880   E(clSetKernelArg(colour, 0, sizeof(cl_mem), &image));
      E(clSetKernelArg(colour, 1, sizeof(cl_mem), &out_rgb));
      E(clSetKernelArg(colour, 2, sizeof(cl_int), &count));
      E(clGetKernelWorkGroupInfo(colour, device_id, CL_KERNEL_WORK_GROUP_SIZE, ↙
          ↳ sizeof(local), &local, NULL));
      global = ((count + local - 1)/local) * local;
885   cl_event coloured;

      // copy image to OpenGL
    #ifdef USE_CLGL
      E(clEnqueueNDRangeKernel(commands, colour, 1, NULL, &global, &local, 1, &↙
          ↳ acquired, &coloured));
890   cl_event released;
      E(clEnqueueReleaseGLObjects(commands, 1, &out_rgb, 1, &coloured, &released));
      clWaitForEvents(1, &released);
      glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, w, h, GL_RGB, GL_UNSIGNED_BYTE, 0);
    #else
895   E(clEnqueueNDRangeKernel(commands, colour, 1, NULL, &global, &local, 1, &↙
          ↳ caniterate, &coloured));
      E(clEnqueueReadBuffer(commands, out_rgb, CL_TRUE, 0, gbytes, rgb, 1, &coloured↙
          ↳ , 0));
      glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, w, h, GL_RGB, GL_UNSIGNED_BYTE, rgb);
    #endif

900   // display image
      glBegin(GL_QUADS); {
    #define V(x,y) glTexCoord2f(x, y);glVertex2f(2*x-1, 2*y-1);
      V(0,0)V(1,0)V(1,1)V(0,1)
    #undef V
905   } glEnd();
      glfwSwapBuffers(window);

      // copy image to memory
      cl_event readrgb;
910   #ifdef USE_CLGL
      glBindTexture(GL_TEXTURE_2D, 0);
      glFinish();
      cl_event acquired2;
      E(clEnqueueAcquireGLObjects(commands, 1, &out_rgb, 0, NULL, &acquired2));
915   E(clEnqueueReadBuffer(commands, out_rgb, CL_TRUE, 0, gbytes, &rgb[0], 1, &↙
          ↳ acquired2, &readrgb));
      E(clEnqueueReleaseGLObjects(commands, 1, &out_rgb, 1, &readrgb, NULL));
      glBindTexture(GL_TEXTURE_2D, texture);
    #else
      E(clEnqueueReadBuffer(commands, out_rgb, CL_TRUE, 0, gbytes, &rgb[0], 0, NULL,↙
          ↳ &readrgb));
920   #endif

      // output PPM on stdout
      mpfr_fprintf(stdout, "P6\n%d %d\n# re = ", w, h);
      mpfr_fprintf(stdout, "# %dx%d\n# re = ", w, h);
```

```
925     mpfr_out_str(stdout, 10, 0, c.real().backend().data(), MPFR_RNDN);
        mpfr_fprintf(stdout, "\n# im = ");
        mpfr_out_str(stdout, 10, 0, c.imag().backend().data(), MPFR_RNDN);
        mpfr_fprintf(stdout, "\n# @    %g    P%d p%d\n# rre ", csize, period, precount↙
            ↳ );
        mpfr_out_str(stdout, 10, 0, refc.real().backend().data(), MPFR_RNDN);
930     mpfr_fprintf(stdout, "\n# rim ");
        mpfr_out_str(stdout, 10, 0, refc.imag().backend().data(), MPFR_RNDN);
        mpfr_fprintf(stdout, "\n255\n");
        fwrite(rgb, gbytes, 1, stdout);
        fflush(stdout);
935
        // clean up
        free(rgb);
        free(pre_zs);
        free(per_zs);
940     delete ref;
        clReleaseMemObject(in_pre_zs);
        clReleaseMemObject(in_per_zs);
        clReleaseMemObject(state[0]);
        clReleaseMemObject(state[1]);
945     clReleaseMemObject(keep);
        clReleaseMemObject(sums[0]);
        clReleaseMemObject(sums[1]);
        clReleaseMemObject(image);
        clReleaseMemObject(out_rgb);
950     clReleaseProgram(program);
        clReleaseKernel(initialize);
        clReleaseKernel(iterate);
        clReleaseKernel(prefixsum);
        clReleaseKernel(permute);
955     clReleaseKernel(colour);
        clReleaseCommandQueue(commands);
        clReleaseContext(context);
        glfwTerminate();
        return 0;
960 }
```

33 mandelbrot-delta-cl/mandelbrot-delta-cl-exp.cc

```
// mandelbrot-delta-cl -- Mandelbrot set perturbation renderer using OpenCL
// Copyright: (C) 2013 Claude Heiland-Allen <claude@mathr.co.uk>
// License: http://www.gnu.org/licenses/gpl.html GPLv3+

5   // some systems list available platforms in different orders
    // some platforms do not support double precision
    // change this line to suit your system's platforms
    // platforms are listed to stderr on startup
    #define PLATFORM 0
10
    #include <complex>
    #include <math.h>
    #include <stdio.h>
    #include <stdlib.h>
15  #include <boost/multiprecision/mpfr.hpp>
    #include <CL/cl.h>
    #include "cl-extra.cc"
```

```
using namespace boost::multiprecision;
typedef mpfr_float R;
typedef std::complex<R> C;
typedef double F53;
typedef std::complex<F53> C53;

// the OpenCL kernel

#define CL(cl) "#pragma OPENCL EXTENSION cl_khr_fp64: require\n" #cl

const char *src = CL(

__kernel void delta
  ( __global double* zs
  , const unsigned int period
  , __global float* output // { normalizeddistanceestimate, continuousdwell, atomperioddomain }
  , const unsigned int w
  , const unsigned int h
  , const unsigned int j
  , const double r
  , const unsigned int chunk
  ) {
  int i = get_global_id(0);
  int x = i % w;
  int y = i / w;
  if (y < chunk) {
    // d0 <- pixel coordinates
    double ii = x;
    double jj = j + y;
    double pixelSpacing = 16.0 * pow(r / 64.0, jj / h) - 16.0 * pow(r / 64.0, (jj+1) / h);
    double d0x = cos(2.0 * 3.141592653 * ii / w) * 16.0 * pow(r / 64.0, jj / h); //(ii - w/2) * r / (h/2);
    double d0y = sin(2.0 * 3.141592653 * ii / w) * 16.0 * pow(r / 64.0, jj / h); //(jj - h/2) * r / (h/2);
    // d <- d0
    double dx = d0x;
    double dy = d0y;
    // dydc <- 1
    double dydcx = 1.0;
    double dydcy = 0.0;
    double er2 = 65536.0;
    float distance = -1.0;
    float dwell = -1.0;
    float atomperiod = 1.0;
    double my2 = er2;
    for (int k = 0; k < 0x1000; ++k) { // FIXME hardcoded maxiters
      int p = k % period;
      double zx = zs[2*p+0];
      double zy = zs[2*p+1];
      // y <- z + d
      double yx = zx + dx;
      double yy = zy + dy;
      double y2 = yx * yx + yy * yy;
      double ly2 = log(y2);
```

```
      if (! (y2 <= er2)) {
        // distance <- |y| log|y| / |dydc|
        double y = sqrt(y2);
        double dydc2 = dydcx * dydcx + dydcy * dydcy;
        double dydc = sqrt(dydc2);
        distance = y * ly2/2.0 / (dydc * pixelSpacing) / 4.0;
        dwell = k - log2(ly2 / log(er2));
        break;
      }
//      ly2 -= 0.5 * (k + 1);
      if (ly2 < my2) {// && (k + 1) % period != 0) {
        my2 = ly2;
        atomperiod = k + 1.0;
      }
      // d <- 2 z d + d d + d0
      double tx = 2.0 * (zx * dx - zy * dy) + (dx * dx - dy * dy) + d0x;
      double ty = 2.0 * (zx * dy + zy * dx) + (2.0 * dx * dy) + d0y;
      // dydc <- 2 y dydc + 1
      double sx = 2.0 * (yx * dydcx - yy * dydcy) + 1.0;
      double sy = 2.0 * (yx * dydcy + yy * dydcx);
      dx = tx;
      dy = ty;
      dydcx = sx;
      dydcy = sy;
    }
    output[3*i+0] = distance;
    output[3*i+1] = dwell;
    output[3*i+2] = atomperiod;
  }
}

);

// the main program

int main(int argc, char **argv) {

  // read command line arguments
  if (argc != 7) {
    fprintf(stderr, "usage: %s width height real imag size period > out.pgm", ↙
        ↳ argv[0]);
    return 1;
  }
  unsigned int w(atoi(argv[1]));
  unsigned int period(atoi(argv[6]));
  F53 r(atof(argv[5]));
  R::default_precision(10 - log10(r));
  C c = C(R(argv[3]), R(argv[4]));
  unsigned int h = w * (-log(r / 64.0) / (2.0 * 3.141592653)); //(atoi(argv[2]))↙
      ↳ ;
  unsigned int chunk = (((1 << 16) - 1) + w) / w;

  // print viewing parameters
  mpfr_fprintf(stderr, "# %dx%d\n# re = ", w, h);
  mpfr_out_str(stderr, 10, 0, c.real().backend().data(), MPFR_RNDN);
  mpfr_fprintf(stderr, "\n# im = ");
  mpfr_out_str(stderr, 10, 0, c.imag().backend().data(), MPFR_RNDN);
```

```
  mpfr_fprintf(stderr, "\n# @    %g\n# P    %d\n", r, period);

  // compute high precision zs and round to F53
  int zbytes = sizeof(F53) * 2 * period;
  F53 *zs = (F53 *) calloc(1, zbytes);
  C z(0.0, 0.0);
  for (unsigned int i = 0; i < period; ++i) {
    z = z * z + c;
    zs[2*i+0] = F53(z.real());
    zs[2*i+1] = F53(z.imag());
  }

  // enumerate OpenCL platforms
  cl_platform_id platform_id[64];
  cl_uint platform_ids;
  E(clGetPlatformIDs(64, &platform_id[0], &platform_ids));
  for (cl_uint i = 0; i < platform_ids; ++i) {
    fprintf(stderr, "platform %d\n", i);
    char buf[1024];
    cl_platform_info info[5] =
      { CL_PLATFORM_PROFILE
      , CL_PLATFORM_VERSION
      , CL_PLATFORM_NAME
      , CL_PLATFORM_VENDOR
      , CL_PLATFORM_EXTENSIONS
      };
    for (int w = 0; w < 5; ++w) {
      E(clGetPlatformInfo(platform_id[i], info[w], 1024, &buf[0], NULL));
      fprintf(stderr, "\t%s\n", buf);
    }
  }

  // create context
  cl_device_id device_id;
  E(clGetDeviceIDs(platform_id[PLATFORM], CL_DEVICE_TYPE_ALL, 1, &device_id, ↙
      ↘ NULL));
  cl_context_properties properties[] =
    { CL_CONTEXT_PLATFORM, (cl_context_properties) platform_id[PLATFORM]
    , 0
    };
  int err;
  cl_context context = clCreateContext(properties, 1, &device_id, NULL, NULL, &↙
      ↘ err);
  if (! context) { E(err); }
  cl_command_queue commands = clCreateCommandQueue(context, device_id, 0, &err);
  if (! commands) { E(err); }

  // compile program
  cl_program program = clCreateProgramWithSource(context, 1, &src, NULL, &err);
  if (! program) { E(err); }
  err = clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
  if (err != CL_SUCCESS) {
    char buf[1024]; buf[0] = 0;
    clGetProgramBuildInfo(program, device_id, CL_PROGRAM_BUILD_LOG, 1024, &buf↙
        ↘ [0], NULL);
    fprintf(stderr, "build failed:\n%s\n", buf);
    E(err);
```

```
180     }
        cl_kernel kernel = clCreateKernel(program, "delta", &err);
        if (! kernel) { E(err); }

        // create buffers
185     unsigned int count = w * chunk;
        cl_mem in_zs = clCreateBuffer(context, CL_MEM_READ_ONLY, zbytes, NULL, NULL);
        if (! in_zs) { exit(1); }
        cl_mem out_output = clCreateBuffer(context, CL_MEM_WRITE_ONLY, 3*count*sizeof(↙
            ↳ float), NULL, NULL);
        if (! out_output) { exit(1); }
190     E(clEnqueueWriteBuffer(commands, in_zs, CL_TRUE, 0, zbytes, &zs[0], 0, NULL, ↙
            ↳ NULL));
        unsigned int obytes = 3 * count * ((h + chunk - 1)/chunk) * sizeof(float);
        float *output = (float *) calloc(1, obytes);

        // set arguments
195     unsigned int j = 0;
        E(clSetKernelArg(kernel, 0, sizeof(cl_mem), &in_zs));
        E(clSetKernelArg(kernel, 1, sizeof(unsigned int), &period));
        E(clSetKernelArg(kernel, 2, sizeof(cl_mem), &out_output));
        E(clSetKernelArg(kernel, 3, sizeof(unsigned int), &w));
200     E(clSetKernelArg(kernel, 4, sizeof(unsigned int), &h));
        E(clSetKernelArg(kernel, 5, sizeof(unsigned int), &j));
        E(clSetKernelArg(kernel, 6, sizeof(double), &r));
        E(clSetKernelArg(kernel, 7, sizeof(unsigned int), &chunk));
        size_t local;
205     E(clGetKernelWorkGroupInfo(kernel, device_id, CL_KERNEL_WORK_GROUP_SIZE, ↙
            ↳ sizeof(local), &local, NULL));
        size_t global = ((count + local - 1)/local) * local;

        // run the kernel for each row
        for (j = 0; j < h; j += chunk) {
210       fprintf(stderr, "%8d\r", j);
          E(clSetKernelArg(kernel, 5, sizeof(unsigned int), &j));
          E(clEnqueueNDRangeKernel(commands, kernel, 1, NULL, &global, &local, 0, NULL↙
              ↳ , NULL));
          E(clEnqueueReadBuffer(commands, out_output, CL_TRUE, 0, 3*w*chunk*sizeof(↙
              ↳ float), &output[3*w*j], 0, NULL, NULL));
        }
215     clFinish(commands);

    /*
        // analyse data
        FILE *fatom = fopen("atom.dat", "wb");
220     for (unsigned int i = 2; i < 3 * w * h; i += 3) {
          fprintf(fatom, "%d\n", int(output[i]));
        }
        fclose(fatom);
    */
225     // convert to image
        unsigned char *rgb = (unsigned char *) calloc(1, 3 * w * h);
    //  unsigned char *grey = (unsigned char *) calloc(1, w * h);
        float pi = 3.141592653;
        float phi = (sqrt(5.0) + 1.0) / 2.0;
230     float modulus = 24.0 + 1.0/(phi*phi);
        for (unsigned int i = 0; i < 3 * w * h; i += 3) {
```

```
//     grey[i/3] = fmin(fmax(255.0 * sqrt(output[i+0]), 0.0), 255.0);
      float value = fmin(fmax(sqrt(output[i+0]), 0.0), 1.0) * 0.5 + 0.125;
      float chroma = 2.0 * pi * fmod(output[i+2], modulus)/modulus;
      float y = value;
      float u = 0.5 * value * cos(chroma);
      float v = 0.5 * value * sin(chroma);
      float r = y + 1.13983 * v;
      float g = y - 0.39465 * u - 0.58060 * v;
      float b = y + 2.03211 * u;
      if (output[i+0] < 0.0 || output[i+0] > w) {
        rgb[i+0] = rgb[i+1] = rgb[i+2] = 255;
      } else {
        rgb[i+0] = fmin(fmax(r * 255.0, 0.0), 255.0);
        rgb[i+1] = fmin(fmax(g * 255.0, 0.0), 255.0);
        rgb[i+2] = fmin(fmax(b * 255.0, 0.0), 255.0);
      }
    }

  /*
    FILE *fdist = fopen("dist.dat", "wb");
    for (unsigned int i = 0; i < 3 * w * h; i += 3) {
      fprintf(fdist, "%e\n", output[i+0]);
    }
    fclose(fdist);
  */
  /*
    FILE *fraw = fopen("out.raw", "wb");
    fwrite(output, obytes, 1, fraw);
    fclose(fraw);
  */
  /*
    FILE *fgrey = fopen("out.pgm", "wb");
    fprintf(fgrey, "P5\n%d %d\n255\n", w, h);
    fwrite(grey, w * h, 1, fgrey);
    fclose(fgrey);
  */
    // output PPM data on stdout
    mpfr_fprintf(stdout, "P6\n%d %d\n# re = ", w, h);
    mpfr_out_str(stdout, 10, 0, c.real().backend().data(), MPFR_RNDN);
    mpfr_fprintf(stdout, "\n# im = ");
    mpfr_out_str(stdout, 10, 0, c.imag().backend().data(), MPFR_RNDN);
    mpfr_fprintf(stdout, "\n# @    %g\n# P    %d\n255\n", r, period);
    fwrite(rgb, 3 * w * h, 1, stdout);

    // clean up
    clReleaseMemObject(in_zs);
    clReleaseMemObject(out_output);
    clReleaseProgram(program);
    clReleaseKernel(kernel);
    clReleaseCommandQueue(commands);
    clReleaseContext(context);

    return 0;
  }
```

34 mandelbrot-delta-cl/README

```
mandelbrot-delta-cl
===================

Mandelbrot set perturbation renderer using OpenCL

Copyright: (C) 2013,2018 Claude Heiland-Allen <claude@mathr.co.uk>
License: http://www.gnu.org/licenses/gpl.html GPLv3+


Usage
-----

./mandelbrot-delta-cl platform width height cre cim csize polyiters maxiters ↙
    ↳ > out.ppm

cre cim csize are the coordinates of the view. maxiters can be set very
high, it bails out early if the rate of escaping pixels is low. polyiters
is the number of iterations to assume the polynomial approximation is valid
for (setting it to 0 is safe, setting it too high might lead to bad images).

There are some configuration options at the top of the main source file,
but the default options should work ok.


Theory
------

Inspired by SuperFractalThing http://www.superfractalthing.co.nf/
Compute a reference point at high precision, use lower precision deltas
against that reference for all other points. Further use a polynomial
approximation to initialize each pixel at the polyiters'th iteration.
Currently uses a degree-12 polynomial.

Using Maxima
------------

Calculating polynomial approximation recurrences can be done by machine,
to collect coefficients of like terms. Renamed variables compared to
the SFT paper: z = X_n, y = delta_n, x = delta_0; additionally w is
the difference in the derivative (X'_n = X_n + y, A'_n = A_n + w) also
approximated by a polynomial in x = d_0.

<maxima> facsum(expand(subst(
  y = a * x + b * x^2 + c * x^3 + d * x^4 + e * x ^ 5 + f * x^6 + g * x^7 + h * ↙
      ↳ x^8 + i * x^9 + j * x^10 + k * x^11 + l * x^12,
  2 * z * y + y^2 + x)),
  x
);

 x^12*(2*l*z+2*a*k+2*b*j+2*c*i+2*d*h+2*e*g+f^2)
 +2*x^11*(k*z+a*j+b*i+c*h+d*g+e*f)+x^10*(2*j*z+2*a*i+2*b*h+2*c*g+2*d*f+e^2)
 +2*x^9*(i*z+a*h+b*g+c*f+d*e)+x^8*(2*h*z+2*a*g+2*b*f+2*c*e+d^2)
 +2*x^7*(g*z+a*f+b*e+c*d)+x^6*(2*f*z+2*a*e+2*b*d+c^2)+2*x^5*(e*z+a*d+b*c)
 +x^4*(2*d*z+2*a*c+b^2)+2*x^3*(c*z+a*b)+x^2*(2*b*z+a^2)+x*(2*a*z+1)+l^2*x^24
 +2*k*l*x^23+(2*j*l+k^2)*x^22+2*(i*l+j*k)*x^21+(2*h*l+2*i*k+j^2)*x^20
 +2*(g*l+h*k+i*j)*x^19+(2*f*l+2*g*k+2*h*j+i^2)*x^18+2*(e*l+f*k+g*j+h*i)*x^17
 +(2*d*l+2*e*k+2*f*j+2*g*i+h^2)*x^16+2*(c*l+d*k+e*j+f*i+g*h)*x^15
```

```
 +(2*b*l+2*c*k+2*d*j+2*e*i+2*f*h+g^2)*x^14+2*(a*l+b*k+c*j+d*i+e*h+f*g)*x^13

<maxima> facsum(expand(subst(
  w = aa * x + bb * x^2 + cc * x^3 + dd * x^4 + ee * x^5 + ff * x^6 + gg * x^7 +↙
      ↘  hh * x^8 + ii * x^9 + jj * x^10 + kk * x^11 + ll * x^12, subst(
  y = a * x + b * x^2 + c * x^3 + d * x^4 + e * x ^ 5 + f * x^6 + g * x^7 + h * ↙
      ↘ x^8 + i * x^9 + j * x^10 + k * x^11 + l * x^12,
  2 * a * y + 2 * w * (z + y)))),
  x
));

 2*x^12*(ll*z+a*l+a*kk+aa*k+b*jj+bb*j+c*ii+cc*i+d*hh+dd*h+e*gg+ee*g+f*ff)
 +2*x^11*(kk*z+a*k+a*jj+aa*j+b*ii+bb*i+c*hh+cc*h+d*gg+dd*g+e*ff+ee*f)
 +2*x^10*(jj*z+a*j+a*ii+aa*i+b*hh+bb*h+c*gg+cc*g+d*ff+dd*f+e*ee)
 +2*x^9*(ii*z+a*i+a*hh+aa*h+b*gg+bb*g+c*ff+cc*f+d*ee+dd*e)
 +2*x^8*(hh*z+a*h+a*gg+aa*g+b*ff+bb*f+c*ee+cc*e+d*dd)
 +2*x^7*(gg*z+a*g+a*ff+aa*f+b*ee+bb*e+c*dd+cc*d)
 +2*x^6*(ff*z+a*f+a*ee+aa*e+b*dd+bb*d+c*cc)
 +2*x^5*(ee*z+a*e+a*dd+aa*d+b*cc+bb*c)+2*x^4*(dd*z+a*d+a*cc+aa*c+b*bb)
 +2*x^3*(cc*z+a*c+a*bb+aa*b)+2*x^2*(bb*z+a*b+a*aa)+2*x*(aa*z+a^2)+2*l*ll*x^24
 +2*(k*ll+kk*l)*x^23+2*(j*ll+jj*l+k*kk)*x^22+2*(i*ll+ii*l+j*kk+jj*k)*x^21
 +2*(h*ll+hh*l+i*kk+ii*k+j*jj)*x^20+2*(g*ll+gg*l+h*kk+hh*k+i*jj+ii*j)*x^19
 +2*(f*ll+ff*l+g*kk+gg*k+h*jj+hh*j+i*ii)*x^18
 +2*(e*ll+ee*l+f*kk+ff*k+g*jj+gg*j+h*ii+hh*i)*x^17
 +2*(d*ll+dd*l+e*kk+ee*k+f*jj+ff*j+g*ii+gg*i+h*hh)*x^16
 +2*(c*ll+cc*l+d*kk+dd*k+e*jj+ee*j+f*ii+ff*i+g*hh+gg*h)*x^15
 +2*(b*ll+bb*l+c*kk+cc*k+d*jj+dd*j+e*ii+ee*i+f*hh+ff*h+g*gg)*x^14
 +2*(a*ll+aa*l+b*kk+bb*k+c*jj+cc*j+d*ii+dd*i+e*hh+ee*h+f*gg+ff*g)*x^13


Tested With
-----------

Debian Wheezy with a few things from Jessie (recent Boost, ...)
NVIDIA GTX-550Ti      GPU / OpenCL 1.1 CUDA 4.2.1
AMD Athlon II X4 640  CPU / OpenCL 1.2 AMD-APP (938.2)
NVIDIA GeForce G105M  GPU / OpenCL 1.1 CUDA 4.2.1
Intel Core2 Duo P7550 CPU / OpenCL 1.2 AMD-APP (938.2)

Debian Buster with Jessie-Backports non-free amd-opencl-icd
AMD Radeon RX 580 GPU / OpenCL 1.1 Mesa 18.0.4 Clover
  real 1m02.608s / user  0m26.120s / sys 0m1.048s
AMD Ryzen 7 2700X CPU / OpenCL 1.2 AMD-APP (1912.5)
  real 1m57.504s / user 22m41.528s / sys 0m4.122s


Future Work
-----------

better error reporting and recovery
interactive operation
compute precount automatically
investigate whether high degree polynomials are worth it
recursive image subdivision


Bugs
```

```
----

sometimes there are some blobs that shouldn't be there
(not caused by iteration limit being reached, mystery..)


--
http://mathr.co.uk
```

35 mandelbrot-delta-cl/render-library.sh

```
#!/bin/bash
platform=0
width=512
height=512
out=sft-library
maxiters=1000000
mkdir -p "${out}"
while read name
do
  read s
  read r
  read i
  read ignore
  echo "${name}"
  time ./mandelbrot-delta-cl "${platform}" "${width}" "${height}" "${r:2}" "${i:2}" "${s:2}" 1 "${maxiters}" > "${out}/${name}.ppm"
done
```

36 mandelbrot-delta-cl/sft-library.txt

```
Blobby
s=3.945027e-59
r=-1.32274937053405530965319737246380146767496120919682467082541437077 05
i=0.11215137792107857749242449374269442912190722464230227685405588576261
iteration_limit=6912
Bloodshot
s=2.150840e-56
r=0.14355621452983444961422555375035989741273570589747906301679965568498
i=0.65207264731080062833688926376479464152101757916253449412791770164159
iteration_limit=4096
Bug
s=1.427670e-138
r=-0.97103220440086297060598774038636585293938436533891946814778295746265231260392183633 38
i=0.26126607402860182049532560145576605876547859935905557411755234391859334703252701910566
iteration_limit=6400
Cell
s=4.247607e-31
r=-0.10805164355127826983860338850618974537 4
i=0.96202118313113025846707817996680626674 22
iteration_limit=25688
Chains
```

```
s=1.124769e-70
r↙
    ↳ =0.059225953558725264703470790955926721025398828980634406426275081985993643350095↙
    ↳
i↙
    ↳ =0.655759481607671167418333236185189577645669410299365700053949615279256727603399↙
    ↳
iteration_limit=35279
Chaos
s=2.078600e-29
r=-0.5316466253832941645290557688141759096
i=0.668530830012130438873108564410163919 28
iteration_limit=8764
coil
s=8.3753E-31
r=-1.039185705117829991613876838902973521 28
i=0.348773723803690742738922183424402127 36
iteration_limit=5878
Coral
s=5.437534e-65
r=-0.207975238753095626766617136501241794405143243672131396312340491439384909
i=1.109158264093554719253087572273577938747779709018260640843031210697634514 8
iteration_limit=16640
Cross
s=1.048582e-41
r=-1.322749370534055309653197372463801467674961485530
i=0.112151377921078577492424493742694429121906525968 5
iteration_limit=4864
Deep
s=5.054594e-264
r↙
    ↳ =-0.863516907933672378790985267355003631711229009266233902356568590290959858174791066678 9
    ↳
i↙
    ↳ =0.247700850855426848979201549411145329785716529125852075911990326054891624344755799016 21
    ↳
iteration_limit=13824
Diamond
s=1.117441e-35
r=-0.139979599473469010707455127696075574742775
i=0.992096596034991738754532113365686599363743 8
iteration_limit=4608
Fans
s=6.66575E-26
r=-1.479734172917307109953944973796048347
i=-0.000330734912098132398970523800624035
iteration_limit=63019
Flower
s=3.957286e-52
r=-0.702178381174589199724289236224420308187094151370677634321 63
i=0.350188994078575685784373023205753300349944168843451225598036
iteration_limit=40000
H
s=1.003089e-43
r=-0.052041277016725785570697283430408241120996453038 21
i=0.880002647049300592795822899476444917140801358994528
iteration_limit=8384
```

```
II
s=6.406352e-70
r↙
    ↳ =0.059225953558725264703470790955926721025398828980634406424909773141061905353101↙
    ↳
i↙
    ↳ =0.655759481607671167418333236185189577645669410299365700053795401428262327401521↙
    ↳
iteration_limit=65174
Jaws
s=4.810315E-26
r=-1.996375168276809430933711379256262298
i=0.000000018880999937177361002751138426 5
iteration_limit=15000
Leaves
s=8.814956e-23
r=-1.747436602706267670454819608780 3
i=0.002091360491191042009027575842 16
iteration_limit=3130
Linear
s=5.619169e-104
r↙
    ↳ =0.411830674436556008296458918970638255570178130045231134862127745036113652495855682255 09
    ↳
i↙
    ↳ =0.138069633274393601008438572967915122418947930458532450609199200933015148731502006121 64
    ↳
iteration_limit=7168
Lines
s=2.877312e-37
r=0.267631092094222042989030536901435635989043689
i=0.003868418790357451575907772520676739621003884
iteration_limit=4352
Mosaic
s=3.096137e-46
r=0.372137738770323258373356630885867793129508737871396193
i=-0.090398245434178161692952411151009819302665482576 5
iteration_limit=15512
Mosaic2
s=2.672520e-67
r↙
    ↳ =0.372137738770323258373356630885867793129508737741343871454145013316342011146500↙
    ↳
i↙
    ↳ =-0.090398245434178161692952411151009819302665482564540025169572392202936760194 24↙
    ↳
iteration_limit=13652
Mosaic3
s=5.135241e-35
r=0.059225953558725264703470790955926720970754 47
i=0.655759481607671167418333236185189572350282 03
iteration_limit=16843
MultiH
s=6.153873e-59
r=-0.150686282301511539046120286522499136234061419498036024489698996420 9
i=1.044000172907917007726285719898243522495207205075955022336038268043 38
iteration_limit=100000
```

```
Octogram
s=1.500000e-27
r=-0.16119324162833354979320377699426 50
i=1.03962741547672189695078100332888192
iteration_limit=5009
Pinwheel
s=9.7855E-49
r=-0.53164662538329416452905576881073001561127920648965532845 2073
i=0.66853083001213043887310856439782112668056145765042941161 19685
iteration_limit=87640
Ripple
s=6.29145E-16
r=0.29631325659167238737315235
i=-0.017170925735399121326943
iteration_limit=46041
Saw Wheel
s=8.497763e-41
r=0.41149469307133465265064840380739746478574726220760
i=-0.16155567616081125110885316936501759561106880249 52
iteration_limit=2304
Scales
s=1.500000e-38
r=-0.013100842061927557374084845856837719995039905 4
i=0.71505414382185825698128995207711460876924722308
iteration_limit=7065
Shells
s=7.501533e-13
r=0.250117332491014475344
i=0.000001918001511940442
iteration_limit=32000
Spirals
s=5.4293E-22
r=-1.250208131537520520329818240511 80
i=-0.00559127510520559732398195501999
iteration_limit=201367
Square
s=1.212211e-25
r=-0.11397980132277245449197801142900 1
i=0.96957714236040552365562012473919 28
iteration_limit=17600
Squares
s=1.569206e-24
r=-0.55689132537433471241027043327 99
i=0.63531080567597055494660396989646
iteration_limit=204608
Starfish
s=1.500000e-47
r=-0.09854958953119926490560879715163132533733179483143094 18
i=0.92469813784454892364548419957293771795418531023091926976
iteration_limit=422304
Swirl
s=1.500000e-103
r↙
    ↳ =-0.82998929490074390782101955568191178347129471944369585853623324386860499343009075841 07
    ↳
i↙
    ↳ =0.20732341190561053765873371726351614094595448302182762775274270526032200044105699135006
```

```
    ↪
iteration_limit=24784
Tentacle
s=5.208408e-63
r=-0.29209780565293593525502946410247541778702771353552277409212398377545719 05
i=0.65902369962977701211090046594765415228793874786420545507471841694969945884
iteration_limit=23609
Tunnel
s=8.6469E-42
r=0.27522000732176228425895258844200966443778996152984
i=-0.008110665117055043096287974382693951478033373155 8
iteration_limit=50000
Twist
s=3.144748e-70
r↙
    ↪ =0.41183067443655600829645891897063825557017813004523113486212774503611361712719852 37↙
    ↪
i↙
    ↪ =0.13806963327439360100843857296791512241894793045853245060919920093301512071634084 92↙
    ↪
iteration_limit=5120
Wave
s=1.185776e-33
r=-0.10091453933399909667486636783772556781141577
i=0.95638693751228669896213293889236213243418842 2
iteration_limit=6400
```

37 mandelbrot-laurent/DM.gnuplot

```
set term pngcairo size 2000,1000 font "LMSans10" fontscale 2
set output "DM.png"
unset key
set grid
set multiplot layout 1,2
set title "Unit disk D with external rays and circles"
plot [-2:2][-2:2] "D-equi.dat" w l, "D-ray.dat" w l
set title "Image under Psi_M(w) truncated to 1000 terms"
plot [-2.5:1.5][-2:2] "M-equi.dat" w l, "M-ray.dat" w l
unset multiplot
```

38 mandelbrot-laurent/Laurent.hs

```
{-# LANGUAGE OverloadedStrings #-}
module Laurent (withLaurent) where

import Control.Monad (forM)
import Data.Text (Text)
import Data.Memo.Sqlite

betaF_ :: ((Integer, Integer) -> IO Rational) -> ((Integer, Integer) -> IO ↙
    ↪ Rational)
betaF_ betaF (n, m)
  | m == 0 = return 1
  | 0 < n && m < nnn = return 0
  | otherwise = do
      a <- betaF (n + 1, m)
```

```
      s <- fmap sum . forM [nnn .. m - nnn] $ \k -> do
        nk <- betaF (n, k)
        nmk <- betaF (n, m - k)
        return $! nk * nmk
      b <- betaF (0, m - nnn)
      return $! (a - s - b) / 2
  where
    nnn = 2 ^ (n + 1) - 1

withLaurent :: ((Integer -> IO Rational) -> IO r) -> IO r
withLaurent act = do
  let Just bft = table "betaF"
  (bclean, betaF) <- memoRec' readShow "betaF.sqlite" bft betaF_
  let b (-1) = return 1
      b m = betaF (0, m + 1)
  r <- act b
  bclean
  return r
```

39 mandelbrot-laurent/Main.hs

```
import Control.Monad (forM)
import Data.Complex (cis, Complex((:+)))
import Data.List (intercalate)
import Data.Ratio (denominator, numerator)
import System.Environment (getArgs)
import System.IO (hFlush, stdout)
import Data.Vector.Unboxed (foldr', forM_, fromListN, generate, Vector)
import qualified Data.Vector.Unboxed as V

import Laurent (withLaurent)

main :: IO ()
main = do
  [sn, sm, sr, sw, ss] <- getArgs
  n <- readIO sn
  m <- readIO sm
  bs <- withLaurent $ forM [-1 .. m]
  let v = fromListN (fromInteger m + 2) $ map fromRational bs
      c = case sw of
        "equi" -> circle n (1 + 0.5 ** read sr)
        "ray" -> ray n (read (replace '/' '%' sr))
      j = case ss of
        "M" -> V.map (jungreis v) c
        "D" -> c
        "B" -> V.map (:+ 0) v
  forM_ j $ \(x:+y) -> putStrLn (show x ++ " " ++ show y)

replace x with (y:ys) | x == y = with : replace x with ys
                      | otherwise = y : replace x with ys
replace _ _ _ = []

circle :: Int -> Double -> Vector (Complex Double)
circle n r = generate n (\i -> fmap (r *) . cis $ 2 * pi * (fromIntegral i + ↙
    ↳ 0.5) / fromIntegral n)

ray :: Int -> Rational -> Vector (Complex Double)
```

```
ray n r = generate n (\i -> fmap ((1 + 0.5 ** (53 * fromIntegral i / ↙
    ↳ fromIntegral n)) *) . cis $ 2 * pi * fromRational r)

horner :: Complex Double -> Vector Double -> Complex Double
horner x = foldr' (\a s -> (a :+ 0) + x * s) 0

jungreis :: Vector Double -> Complex Double -> Complex Double
jungreis u w = w * horner (recip w) u
```

40 mandelbrot-series-approximation/args.h

```
#ifndef ARGS_H
#define ARGS_H 1

#include <iostream>
#include <cstdlib>
#include <cstring>
using namespace std;

typedef int N;

// packed arguments
struct arguments
{
  N width;
  N height;
  N maxiters;
  N order;
  N stop;
  N accuracy;
  N precision;
};

// pack arguments
inline char *pack(size_t &size, N width, N height, N maxiters, N order, N stop, ↙
    ↳ N accuracy, N precision, const char *sre, const char *sim, const char *↙
    ↳ sradius, const char *filename)
{
  size = sizeof(arguments) + strlen(sre) + 1 + strlen(sim) + 1 + strlen(sradius)↙
      ↳ + 1 + strlen(filename) + 1;
  arguments *args = (arguments *) malloc(size);
  args->width = width;
  args->height = height;
  args->maxiters = maxiters;
  args->order = order;
  args->stop = stop;
  args->accuracy = accuracy;
  args->precision = precision;
  char *data = (char *) (args + 1);
  memset(data, 0, size - sizeof(arguments));
  strcat(data, sre);
  data += strlen(data) + 1;
  strcat(data, sim);
  data += strlen(data) + 1;
  strcat(data, sradius);
  data += strlen(data) + 1;
  strcat(data, filename);
```

```
  return (char *) args;
}

// unpack arguments
inline void unpack(char *data, size_t size, N &width, N &height, N &maxiters, N &order, N &stop, N &accuracy, N &precision, const char *&sre, const char *&sim, const char *&sradius, const char *&filename)
{
  (void) size;
  arguments *args = (arguments *) data;
  width = args->width;
  height = args->height;
  maxiters = args->maxiters;
  order = args->order;
  stop = args->stop;
  accuracy = args->accuracy;
  precision = args->precision;
  data += sizeof(arguments);
  sre = data;
  data += strlen(data) + 1;
  sim = data;
  data += strlen(data) + 1;
  sradius = data;
  data += strlen(data) + 1;
  filename = data;
}

// parse arguments
inline N parse(int argc, char **argv, N &width, N &height, N &maxiters, N &order, N &stop, N &accuracy, N &precision, const char *&sre, const char *&sim, const char *&sradius, const char *&filename)
{
  // parse arguments
  for (N a = 1; a < argc; ++a)
  {
    if (!strcmp("--width", argv[a]) && a + 1 < argc)
      width = atoi(argv[++a]);
    else
    if (!strcmp("--height", argv[a]) && a + 1 < argc)
      height = atoi(argv[++a]);
    else
    if (!strcmp("--maxiters", argv[a]) && a + 1 < argc)
      maxiters = atoi(argv[++a]);
    else
    if (!strcmp("--order", argv[a]) && a + 1 < argc)
      order = atoi(argv[++a]);
    else
    if (!strcmp("--stopping", argv[a]) && a + 1 < argc)
    {
      N b(++a);
      if (!strcmp("knighty", argv[b]))
        stop = 0;
      else
      if (!strcmp("quaz0r", argv[b]))
        stop = 1;
      else
      if (!strcmp("knighty2", argv[b]))
```

```
          stop = 2;
        else
        {
          cerr << argv[0] << ": bad stopping: " << argv[b] << endl;
          return 1;
        }
      }
      else
      if (!strcmp("--accuracy", argv[a]) && a + 1 < argc)
        accuracy = atoi(argv[++a]);
      else
      if (!strcmp("--precision", argv[a]) && a + 1 < argc)
        precision = atoi(argv[++a]);
      else
      if (!strcmp("--re", argv[a]) && a + 1 < argc)
        sre = argv[++a];
      else
      if (!strcmp("--im", argv[a]) && a + 1 < argc)
        sim = argv[++a];
      else
      if (!strcmp("--radius", argv[a]) && a + 1 < argc)
        sradius = argv[++a];
      else
      if (!strcmp("--output", argv[a]) && a + 1 < argc)
        filename = argv[++a];
      else
      {
        cerr << argv[0] << ": unexpected: " << argv[a] << endl;
        cerr
          << "usage:" << endl
          << "  --width n" << endl
          << "  --height n" << endl
          << "  --maxiters n" << endl
          << "  --order n" << endl
          << "  --stopping knighty|quaz0r|knighty2" << endl
          << "  --accuracy n" << endl
          << "  --precision n" << endl
          << "  --re f" << endl
          << "  --im f" << endl
          << "  --radius f" << endl
          << "  --output s.ppm" << endl
          ;
        return 1;
      }
    }
    return 0;
}

#endif
```

41 mandelbrot-series-approximation/emscripten.cpp

```
#ifndef __EMSCRIPTEN__
#error this file needs to be compiled by emscripten
#endif

#include <SDL/SDL.h>
```

```
#include <emscripten.h>

#include "args.h"

struct main_context
{
  N width;
  N height;
  SDL_Surface *screen;
};
main_context the_main_context;

// callback with results from web worker
void display(char *data, int size, void *arg)
{
  uint8_t *rgb = (uint8_t *) data;
  if (rgb[size - 1] == 255)
  {
    // final response with image data
    main_context *context = (main_context *) arg;
    if (SDL_MUSTLOCK(context->screen)) SDL_LockSurface(context->screen);
    for (N y = 0; y < context->height; ++y)
      for (N x = 0; x < context->width; ++x)
      {
        N k = (y * context->width + x) * 3;
        *((Uint32*)context->screen->pixels + context->width * y + x) =
          SDL_MapRGBA(context->screen->format, rgb[k+0], rgb[k+1], rgb[k+2], ↙
              ↳ 255);
      }
    if (SDL_MUSTLOCK(context->screen)) SDL_UnlockSurface(context->screen);
    SDL_Flip(context->screen);
  }
  else
  {
    // provisional response with logging data
    printf("%s", data);
  }
}

// entry point
extern "C" int main(int argc, char **argv)
{
  // initial defaults
  N width = 640;
  N height = 360;
  N maxiters = 1 << 8;
  N order = 16;
  N stop = 0;
  N accuracy = 24;
  N precision = 0; // auto
  const char *sre = "-0.75";
  const char *sim = "0.0";
  const char *sradius = "1.5";
  const char *filename = "out.ppm";
  // parse arguments
  if (parse(argc, argv, width, height, maxiters, order, stop, accuracy, ↙
      ↳ precision, sre, sim, sradius, filename))
```

```
    return 1;
  // create worker thread
  worker_handle worker = emscripten_create_worker("worker.js");
  // prepare video
  SDL_Init(SDL_INIT_VIDEO);
  the_main_context.width = width;
  the_main_context.height = height;
  the_main_context.screen = SDL_SetVideoMode(width, height, 32, SDL_SWSURFACE);
  // render image in web worker background thread
  size_t size;
  char *data = pack(size, width, height, maxiters, order, stop, accuracy,
      precision, sre, sim, sradius, filename);
  emscripten_call_worker(worker, "render", data, size, display, &
      the_main_context);
  return 0;
}
```

42 mandelbrot-series-approximation/index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Mandelbrot</title>
  </head>
  <body>
    <form action="m.html" method="get">
      <h1>Mandelbrot</h1>
      <h2>Image</h2>
      <h3>Width</h3>
      <input name="width" type="text" size="10" value="1024">
      <h3>Height</h3>
      <input name="height" type="text" size="10" value="576">
      <h2>View</h2>
      <h3>Real Coordinate</h3>
      <textarea name="re" rows="8" cols
          ="100">-1.99999999913827011876648672637779908023155217777256998965597443321632828
          textarea>
      <h3>Imaginary Coordinate</h3>
      <textarea name="im" rows="8" cols
          ="100">5.47376553292304062442712374780046094070664886945802954270821214086480490
          e-20</textarea>
      <h3>View Radius</h3>
      <input name="radius" type="text" size="10" value="5e-900">
      <h3>Maximum Iteration Count</h3>
      <input name="maxiters" type="text" size="10" value="400000">
      <h2>Series Approximation</h2>
      <h3>Order (Number Of Terms)</h3>
      <input name="order" type="text" size="10" value="16">
      <h3>Stopping Condition</h3>
      <select name="stopping">
        <option value="knighty" selected>knighty</option>
        <option value="quaz0r">quaz0r</option>
        <option value="knighty2">knighty2</option>
      </select>
      <h3>Accuracy</h3>
      <input name="accuracy" type="text" size="10" value="24">
      <h2>Render</h2>
```

```
35        <input type="submit" value="Render">
      </form>
    </body>
</html>
```

43 mandelbrot-series-approximation/Makefile

```
m: m.cpp args.h
        g++ -std=c++11 -Wall -Wextra -pedantic -O3 -fopenmp -o m m.cpp -lmpfr
```

44 mandelbrot-series-approximation/Makefile.emscripten

```
prefix ?= $(HOME)/opt

all: m.html worker.js

m.html: emscripten.cpp args.h pre.js
        emcc -std=c++11 -Wall -Wextra -pedantic -O2 -o m.html emscripten.cpp --↙
            ↪ pre-js pre.js

worker.js: m.cpp args.h
        emcc -std=c++11 -Wall -Wextra -pedantic -O2 -o worker.js m.cpp -I$(↙
            ↪ prefix)/include $(prefix)/lib/libmpfr.a $(prefix)/lib/libgmp.a -s ↙
            ↪ 'BUILD_AS_WORKER=1' -s 'ASSERTIONS=1' -g
```

45 mandelbrot-series-approximation/m.cpp

```
// m.cpp 2016,2017 Claude Heiland-Allen
// LICENSE: public domain, cc0, whatever
// this is demonstration code based on algorithms found in fractalforums.com
//
// perturbation technique: mrflay
// * http://www.fractalforums.com/announcements-and-news/superfractalthing-↙
    ↪ arbitrary-precision-mandelbrot-set-rendering-in-java/msg59980/#msg59980
// * http://www.superfractalthing.co.nf/sft_maths.pdf
// glitch detection method: Pauldelbrot
// * http://www.fractalforums.com/announcements-and-news/pertubation-theory-↙
    ↪ glitches-improvement/msg73027/#msg73027
// series stopping condition: knighty
// * http://www.fractalforums.com/announcements-and-news/*continued*-↙
    ↪ superfractalthing-arbitrary-precision-mandelbrot-set-rendering-in-ja/↙
    ↪ msg95532/#msg95532
// * http://www.fractalforums.com/announcements-and-news/*continued*-↙
    ↪ superfractalthing-arbitrary-precision-mandelbrot-set-rendering-in-ja/↙
    ↪ msg95707/#msg95707
// series stopping condition: quaz0r
// * http://www.fractalforums.com/announcements-and-news/*continued*-↙
    ↪ superfractalthing-arbitrary-precision-mandelbrot-set-rendering-in-ja/↙
    ↪ msg95838/#msg95838
// * http://www.fractalforums.com/announcements-and-news/*continued*-↙
    ↪ superfractalthing-arbitrary-precision-mandelbrot-set-rendering-in-ja/↙
    ↪ msg96245/#msg96245
// series stopping condition: knighty2
// * http://www.fractalforums.com/announcements-and-news/*continued*-↙
    ↪ superfractalthing-arbitrary-precision-mandelbrot-set-rendering-in-ja/↙
    ↪ msg96343/#msg96343
```

```
// glitch correction new references: claude
// * http://www.fractalforums.com/announcements-and-news/pertubation-theory-glitches-improvement/msg84154/#msg84154
// * http://www.fractalforums.com/mandelbrot-and-julia-set/calculating-new-reference-points-using-information-from-glitch-detection/msg86138/#msg86138
//
// compile:
//  g++ -std=c++11 -O3 -march=native m.cpp -lmpfr -Wall -Wextra -pedantic -fopenmp
// run:
//  ./a.out --help
// view results:
//  display out.ppm

#include <cassert>
#include <cmath>
#include <cstdlib>
#include <cstring>

#include <algorithm>
#include <complex>
#include <iostream>
#include <vector>

#include <boost/multiprecision/mpfr.hpp>
#include <mpfr.h>

#ifdef __EMSCRIPTEN__
#include <SDL/SDL.h>
#include <emscripten.h>
#define LOG(x) \
do { \
  ostringstream cout; \
  x << endl; \
  emscripten_worker_respond_provisionally(const_cast<char *>(cout.str().c_str()), strlen(cout.str().c_str()) + 1); \
} while(0)
#else
#define LOG(x) x
#endif

#include "args.h"

using namespace std;
using namespace boost;
using namespace multiprecision;

inline double sign(double x) {
  return (x > 0) - (x < 0);
}

class edouble {
private:
  static const int64_t maxexponent = (1LL << 60) - 2;
  static const int64_t minexponent = 2 - (1LL << 60);
  static const int64_t supexponent = maxexponent - 2000;
```

```
70    static const int64_t infexponent = minexponent + 2080;
      double x;
      long e;
    public:
      inline edouble() : x(0), e(0) { };
75    inline edouble(const edouble &that) : x(that.x), e(that.e) { };
      inline edouble(const double &x0, const int64_t &e0) {
        if (x0 == 0 || std::isnan(x0) || std::isinf(x0)) {
          x = x0;
          e = 0;
80      } else {
          int tmp(0);
          double x1(std::frexp(x0, &tmp));
          int64_t e1(tmp);
          e1 += e0;
85        if (e0 > supexponent || e1 > maxexponent) {
            x = sign(x0) / 0.0;
            e = 0;
          } else if (e0 < infexponent || e1 < minexponent) {
            x = sign(x0) * 0.0;
90          e = 0;
          } else {
            x = x1;
            e = e1;
          }
95      }
      };
      inline edouble(const long double &x0, const int64_t &e0) {
        if (x0 == 0 || std::isnan(x0) || std::isinf(x0)) {
          x = x0;
100       e = 0;
        } else {
          int tmp(0);
          long double x1(frexp(x0, &tmp));
          int64_t e1(tmp);
105       e1 += e0;
          if (e0 > supexponent || e1 > maxexponent) {
            x = sign(x0) / 0.0;
            e = 0;
          } else if (e0 < infexponent || e1 < minexponent) {
110         x = sign(x0) * 0.0;
            e = 0;
          } else {
            x = x1;
            e = e1;
115       }
        }
      };
      inline edouble(const double &x) : edouble(x, 0) { };
      inline edouble(const long double &x) : edouble(x, 0) { };
120   inline explicit edouble(const mpfr_t &a) {
        long e(0);
        double x(mpfr_get_d_2exp(&e, a, MPFR_RNDN));
        *this = edouble(x, e);
      }
125   inline edouble(const int &x) : edouble(double(x)) { };
      inline ~edouble() { };
```

```
  inline int64_t exponent() const { return e; };
  inline void to_mpfr(mpfr_t &m) {
    mpfr_set_d(m, x, MPFR_RNDN);
    mpfr_mul_2si(m, m, e, MPFR_RNDN);
  };
  inline long double to_ld() {
    int tmp(e);
    if (e > int64_t(tmp)) {
      return sign(x) / 0.0;
    }
    if (e < int64_t(tmp)) {
      return sign(x) * 0.0;
    }
    return std::ldexp((long double) x, tmp);
  }
  friend inline bool operator==(const edouble &a, const edouble &b);
  friend inline bool operator!=(const edouble &a, const edouble &b);
  friend inline int compare(const edouble &a, const edouble &b);
  friend inline int sign(const edouble &a);
  friend inline edouble abs(const edouble &a);
  friend inline edouble sqrt(const edouble &a);
  friend inline edouble operator+(const edouble &a, const edouble &b);
  friend inline edouble operator-(const edouble &a);
  friend inline edouble operator-(const edouble &a, const edouble &b);
  friend inline edouble operator*(const edouble &a, const edouble &b);
  friend inline edouble recip(const edouble &a);
  friend inline edouble operator/(const edouble &a, const edouble &b);
  friend inline bool isnan(const edouble &a);
  friend inline bool isinf(const edouble &a);
  friend inline edouble ldexp(const edouble &a, int64_t e);
  friend inline std::ostream& operator<<(std::ostream& o, const edouble& a);
  friend inline std::istream& operator>>(std::istream& i, edouble& a);
  friend inline mpfr_float r_hi(const edouble &z);
  friend inline edouble copysign(const edouble &a, const edouble &b);
  friend inline edouble hypot(const edouble &a, const edouble &b);
  friend inline edouble scalbn(const edouble &a, int b);
  friend inline edouble fabs(const edouble &a);
  friend inline edouble fmax(const edouble &a, const edouble &b);
  friend inline edouble logb(const edouble &a);
  explicit inline operator int() { return this->to_ld(); }
  explicit inline operator double() { return this->to_ld(); }
  inline edouble &operator+=(const edouble &a) {
    *this = *this + a;
    return *this;
  };
  inline edouble &operator-=(const edouble &a) {
    *this = *this - a;
    return *this;
  };
  inline edouble &operator*=(const edouble &a) {
    *this = *this * a;
    return *this;
  };
  inline edouble &operator/=(const edouble &a) {
    *this = *this / a;
    return *this;
  };
```

```
};

inline bool operator==(const edouble &a, const edouble &b) {
  return a.e == b.e && a.x == b.x;
}

inline bool operator!=(const edouble &a, const edouble &b) {
  return a.e != b.e || a.x != b.x;
}

inline int compare(const double &a, const double &b) {
  return sign(a - b);
}

inline int compare(const edouble &a, const edouble &b) {
  if (a.x == 0.0) {
    return -sign(b.x);
  }
  if (b.x == 0.0) {
    return sign(a.x);
  }
  int64_t e(std::max(a.e, b.e));
  int64_t da(a.e - e);
  int64_t db(b.e - e);
  int ia(da);
  int ib(db);
  if (int64_t(ia) != da) {
    // a -> 0
    return -sign(b.x);
  }
  if (int64_t(ib) != db) {
    // b -> 0
    return sign(a.x);
  }
  return compare(std::ldexp(a.x, ia), std::ldexp(b.x, ib));
}

inline bool operator<(const edouble &a, const edouble &b) {
  return compare(a, b) < 0;
}

inline bool operator<=(const edouble &a, const edouble &b) {
  return compare(a, b) <= 0;
}

inline bool operator>(const edouble &a, const edouble &b) {
  return compare(a, b) > 0;
}

inline bool operator>=(const edouble &a, const edouble &b) {
  return compare(a, b) >= 0;
}

inline int sign(const edouble &a) {
  return sign(a.x);
}
```

```
inline edouble abs(const edouble &a) {
  return { std::abs(a.x), a.e };
}

inline edouble sqrt(const edouble &a) {
  return { std::sqrt((a.e & 1) ? 2.0 * a.x : a.x), (a.e & 1) ? (a.e - 1) / 2 : a.e / 2 };
}

inline edouble operator+(const edouble &a, const edouble &b) {
  if (a.x == 0.0) {
    return b;
  }
  if (b.x == 0.0) {
    return a;
  }
  int64_t e(std::max(a.e, b.e));
  int64_t da(a.e - e);
  int64_t db(b.e - e);
  int ia(da);
  int ib(db);
  if (int64_t(ia) != da) {
    // a -> 0
    return b;
  }
  if (int64_t(ib) != db) {
    // b -> 0
    return a;
  }
  return edouble(std::ldexp(a.x, ia) + std::ldexp(b.x, ib), e);
}

inline edouble operator-(const edouble &a) {
  return { -a.x, a.e };
}

inline edouble operator-(const edouble &a, const edouble &b) {
  if (a.x == 0.0) {
    return -b;
  }
  if (b.x == 0.0) {
    return a;
  }
  int64_t e(std::max(a.e, b.e));
  int64_t da(a.e - e);
  int64_t db(b.e - e);
  int ia(da);
  int ib(db);
  if (int64_t(ia) != da) {
    // a -> 0
    return -b;
  }
  if (int64_t(ib) != db) {
    // b -> 0
    return a;
  }
  return edouble(std::ldexp(a.x, ia) - std::ldexp(b.x, ib), e);
```

```
}

inline edouble operator*(const edouble &a, const edouble &b) {
  return edouble(a.x * b.x, a.e + b.e);
}

inline double recip(const double &a) {
  return 1.0 / a;
}

inline edouble recip(const edouble &a) {
  return edouble(recip(a.x), -a.e);
}

inline edouble operator/(const edouble &a, const edouble &b) {
  return edouble(a.x / b.x, a.e - b.e);
}

inline bool isnan(const edouble &a) {
  return std::isnan(a.x);
}

inline bool isinf(const edouble &a) {
  return std::isinf(a.x);
}

inline edouble ldexp(const edouble &a, int64_t e) {
  return edouble(a.x, a.e + e);
}

inline void to_mpfr(float from, mpfr_t &to) {
  mpfr_set_flt(to, from, MPFR_RNDN);
}

inline void to_mpfr(double from, mpfr_t &to) {
  mpfr_set_d(to, from, MPFR_RNDN);
}

inline void to_mpfr(long double from, mpfr_t &to) {
  mpfr_set_ld(to, from, MPFR_RNDN);
}

inline void to_mpfr(edouble from, mpfr_t &to) {
  from.to_mpfr(to);
}

inline long double to_ld(float x) {
  return x;
}

inline long double to_ld(double x) {
  return x;
}

inline long double to_ld(long double x) {
  return x;
}
```

```
inline long double to_ld(edouble x) {
  return x.to_ld();
}

template <typename S, typename T>
inline T to_R(S x, const T &dummy) {
  (void) dummy;
  return T(to_ld(x));
}

inline edouble to_R(edouble x, const edouble &dummy) {
  (void) dummy;
  return x;
}

template <typename S, typename T>
inline std::complex<T> to_C(std::complex<S> x, const T &dummy) {
  return std::complex<T>(to_R(std::real(x), dummy), to_R(std::imag(x), dummy));
}

inline std::complex<edouble> to_C(std::complex<edouble> x, const edouble &dummy)↙
    ↳ {
  (void) dummy;
  return x;
}

inline std::ostream& operator<<(std::ostream& o, const edouble& a) {
  return o << a.x << " " << a.e;
}

inline std::istream& operator>>(std::istream& i, edouble& a) {
  double x;
  long e;
  i >> x >> e;
  a = edouble(x, e);
  return i;
}

inline int64_t exponent(float z)
{
  int e;
  frexp(z, &e);
  return e;
}

inline int64_t exponent(double z)
{
  int e;
  frexp(z, &e);
  return e;
}

inline int64_t exponent(long double z)
{
  int e;
  frexp(z, &e);
```

```
410   return e;
    }

    inline int64_t exponent(edouble z)
    {
415   return z.exponent();
    }

    inline int64_t exponent(const mpfr_t z)
    {
420   if (mpfr_regular_p(z))
        return mpfr_get_exp(z);
      return 0;
    }

425 inline bool isfinite(edouble z)
    {
      return !(isinf(z) || isnan(z));
    }

430 inline edouble copysign(const edouble &a, const edouble &b)
    {
      return edouble(copysign(a.x, b.x), a.e);
    }

435 inline edouble hypot(const edouble &a, const edouble &b)
    {
      return sqrt(a * a + b * b);
    }

440 inline edouble scalbn(const edouble &a, int b)
    {
      return edouble(a.x, a.e + b);
    }

445 inline edouble fabs(const edouble &a)
    {
      return edouble(fabs(a.x), a.e);
    }

450 inline edouble fmax(const edouble &a, const edouble &b)
    {
      if (compare(a, b) >= 0)
        return a;
      else
455     return b;
    }

    inline edouble logb(const edouble &a)
    {
460   if (isnan(a))
        return a;
      if (isinf(a))
        return fabs(a);
      if (a.x == 0.0)
465     return edouble(double(edouble::minexponent));
      return logb(a.x) + a.e;
```

```
}

// type aliases
typedef int N;
typedef edouble R_lo;
typedef mpfr_float R_hi;
typedef complex<R_lo> C_lo;
typedef complex<R_hi> C_hi;

inline R_lo r_lo(const R_hi &z)
{
  return R_lo(z.backend().data());
}

inline R_lo r_lo(const char *s)
{
  unsigned p = mpfr_float::default_precision();
  mpfr_float::default_precision(20u);
  R_lo r(R_hi(s).backend().data());
  mpfr_float::default_precision(p);
  return r;
}

inline R_hi r_hi(const R_lo &z)
{
  R_hi w;
  to_mpfr(z, w.backend().data());
  return w;
}

inline C_lo c_lo(const C_hi &z)
{
  return C_lo(r_lo(real(z)), r_lo(imag(z)));
}

inline C_hi c_hi(const C_lo &z)
{
  return C_hi(r_hi(real(z)), r_hi(imag(z)));
}

inline bool isfinite(const R_hi &z)
{
  return mpfr_number_p(z.backend().data());
}

// helpers for Jordan curve method

template<typename T>
N sgn(T a)
{
  if (a < T(0))
    return -1;
  if (a > T(0))
    return  1;
  return 0;
}
```

```
template<typename R>
inline R cross(const complex<R> &a, const complex<R> &b)
{
  return imag(a) * real(b) - real(a) * imag(b);
}

template<typename R>
bool crosses_positive_real_axis(const complex<R> &a, const complex<R> &b)
{
  if (sgn(imag(a)) != sgn(imag(b)))
  {
    complex<R> d(b - a);
    N s(sgn(imag(d)));
    N t(sgn(cross(d, a)));
    return s == t;
  }
  return false;
}

template<typename R>
bool surrounds_origin(const complex<R> &a, const complex<R> &b, const complex<R>
    &c, const complex<R> &d)
{
  return 1 == (1 &
    ( crosses_positive_real_axis(a, b)
    + crosses_positive_real_axis(b, c)
    + crosses_positive_real_axis(c, d)
    + crosses_positive_real_axis(d, a)
    ));
}

// Jordan curve method for finding lowest period atom in a region
class boxperiod
{
public:
  N n;
  C_hi c[4];
  C_hi z[4];

  boxperiod(C_hi c0, R_lo r)
  : n(1)
  {
    c[0] = c0 + C_hi(r_hi( r), r_hi( r));
    c[1] = c0 + C_hi(r_hi(-r), r_hi( r));
    c[2] = c0 + C_hi(r_hi(-r), r_hi(-r));
    c[3] = c0 + C_hi(r_hi( r), r_hi(-r));
    z[0] = c[0];
    z[1] = c[1];
    z[2] = c[2];
    z[3] = c[3];
  };

  N period(N maxsteps)
  {
    LOG(cout << "computing period" << endl);
    for (N k = 0; k < maxsteps; ++k)
    {
```

```
580         if (n % 1000 == 0)
              LOG(cout << "\rreference period " << n);
            if (surrounds_origin(z[0], z[1], z[2], z[3]))
            {
              LOG(cout << "\rreference period " << n << " (complete)" << endl);
585           return n;
            }
            bool escaped(false);
            for (N j = 0; j < 4; ++j)
            {
590           z[j] = z[j] * z[j] + c[j];
              escaped |= norm(z[j]) > R_hi(1 << 24);
            }
           ++n;
            if (escaped)
595           return 0;
        }
        return 0;
      };
    };
600
    // find the last partial
    N last_partial(C_hi c, N maxiters)
    {
      C_hi z(c);
605   R_hi zmin(norm(z));
      N partial(1);
      LOG(cout << "computing period (try 2)" << endl);
      for (N k = 2; k < maxiters + 2; ++k)
      {
610     if (k % 1000 == 0)
          LOG(cout << "\rreference period (try 2) " << partial << " " << k);
        z = z * z + c;
        R_hi zn(norm(z));
        if (zn > R_hi(4))
615       break;
        if (zn < zmin)
        {
          zmin = zn;
          partial = k;
620     }
      }
      LOG(cout << "\rreference period (try 2) " << partial << " (complete)" << endl)↙
          ↳ ;
      return partial;
    }
625
    // Newton's method for nucleus

    template<typename R>
    bool isfinite(const complex<R> &a)
630 {
      return isfinite(real(a)) && isfinite(imag(a));
    }

    C_hi newton_nucleus(C_hi guess, N period, N maxsteps)
635 {
```

```
  C_hi c(guess);
  LOG(cout << "computing nucleus" << endl);
  N k;
  for (k = 0; k < maxsteps; ++k)
  {
    C_hi z(0, 0);
    C_hi dc(1, 0);
    for (N p = 0; p < period; ++p)
    {
      dc = R_hi(2) * dc * z + C_hi(1, 0);
      z = z * z + c;
      if (p % 1000 == 0)
        LOG(cout << "\rreference nucleus " << k << "/" << maxsteps << " " << p <<
            << "/" << period);
    }
    C_hi cnext(c - z / dc);
    if (isfinite(cnext))
      c = cnext;
    else
      break;
  }
  LOG(cout << "\rreference nucleus " << k << "/" << maxsteps << " (complete)" <<
      endl);
  return c;
}

// reference orbit for perturbation rendering
class reference
{
public:
  N n0;
  N n;
  C_hi z;
  C_hi c;
  vector<C_lo> z_ref;
  vector<R_lo> z_size;  // scaled absolute value for glitch detection

  // constructor
  reference(N n, C_hi z, C_hi c)
  : n0(n)
  , n(n)
  , z(z)
  , c(c)
  {
    C_lo z_lo(c_lo(z));
    z_ref.push_back(z_lo);
    z_size.push_back(1.0e-6 * norm(z_lo));
  };

  // step a reference one iteration
  // returns false if reference escaped
  bool step()
  {
    n = n + 1;
    z = z * z + c;
    C_lo z_lo(c_lo(z));
    z_ref.push_back(z_lo);
```

```
    z_size.push_back(1.0e-6 * norm(z_lo));
    return (norm(z_lo) <= 4);
  };

  // step a reference until it escapes or maxiters reached
  // return false if escaped
  bool run(N maxiters)
  {
    LOG(cout << "computing reference" << endl);
    while (n < maxiters && step())
      if (n % 1000 == 0)
        LOG(cout << "\rreference iteration " << n);
    LOG(cout << "\rreference iteration " << n << " (complete)" << endl);
    return n == maxiters;
  }
};

// reference orbit for perturbation rendering with derivatives for interior de
class reference_interior
{
public:
  N p;
  N n;
  C_hi c;
  C_hi z;
  C_hi dz;
  C_hi dc;
  C_hi dzdz;
  C_hi dcdz;
  vector<R_lo> z_size;  // scaled absolute value for glitch detection
  vector<C_lo> z_ref;
  vector<C_lo> dz_ref;
  vector<C_lo> dc_ref;
  vector<C_lo> dzdz_ref;
  vector<C_lo> dcdz_ref;

  // constructor
  reference_interior(N p, C_hi c) // c must be a cardioid nucleus of period p
  : p(p)
  , n(0)
  , c(c)
  , z(0)
  , dz(1)
  , dc(0)
  , dzdz(0)
  , dcdz(0)
  {
    C_lo z_lo(c_lo(z));
    z_size.push_back(1.0e-6 * norm(z_lo));
    z_ref.push_back(z_lo);
    dz_ref.push_back(c_lo(dz));
    dc_ref.push_back(c_lo(dc));
    dzdz_ref.push_back(c_lo(dzdz));
    dcdz_ref.push_back(c_lo(dcdz));
  };

  // step a reference one iteration
```

```
    // returns false if reference escaped
    bool step()
    {
      n = n + 1;
      dcdz = 2 * z * dcdz + 2 * dc * dz;
      dzdz = 2 * z * dzdz + 2 * dz * dz;
      dc = 2 * z * dc + 1;
      dz = 2 * z * dz;
      z = z * z + c;
      C_lo z_lo(c_lo(z));
      z_size.push_back(1.0e-6 * norm(z_lo));
      z_ref.push_back(z_lo);
      dz_ref.push_back(c_lo(dz));
      dc_ref.push_back(c_lo(dc));
      dzdz_ref.push_back(c_lo(dzdz));
      dcdz_ref.push_back(c_lo(dcdz));
      return (norm(z_lo) <= 4);
    };

    // step a reference until it escapes or maxiters reached
    // return false if escaped
    bool run(N maxiters)
    {
      while (n < maxiters && step())
        cerr << "\rreference iteration " << n;
      cerr << endl;
      return n == maxiters;
    }
  };

  // perturbation technique per-pixel iterations with
  // Pauldelbrot's glitch detection heuristic
  class perturbation
  {
  public:
    N x;
    N y;
    N n;
    C_lo dc;
    C_lo dz;
    C_lo dzdc;
    bool glitched;
    bool escaped;

    // constructor initializes
    perturbation(N x, N y, N n, C_lo dc, C_lo dz, C_lo dzdc)
    : x(x)
    , y(y)
    , n(n)
    , dc(dc)
    , dz(dz)
    , dzdc(dzdc)
    , glitched(false)
    , escaped(false)
    { };

    // calculate pixel
```

```
805     void run(const reference &r, N maxiters)
        {
          while (n < maxiters)
          {
            C_lo z(dz + r.z_ref[n - r.n0]);
810         R_lo nz(norm(z));
            if (nz < r.z_size[n - r.n0])
            {
              glitched = true;
              dz = z;
815           break;
            }
            if (nz > R_lo(65536))
            {
              escaped = true;
820           dz = z;
              break;
            }
            dzdc = R_lo(2) * z * dzdc + C_lo(1);
            dz = R_lo(2) * r.z_ref[n - r.n0] * dz + dz * dz + dc;
825         n = n + 1;
          }
        };
      };

830   // series approximation with various stopping conditions
      class series_approximation
      {
      public:
        enum stopping { knighty = 0, quaz0r = 1, knighty2 = 2 };
835
        N n;              // iteration count
        R_lo dt;          // pixel size
        R_lo tmax;        // view size
        C_hi z;           // reference z at high precision
840     C_hi c;           // reference c at high precision
        N m;              // series order
        vector<C_lo> a;   // coefficients (including z at start and R at end)
        vector<C_lo> b;   // derivative coefficients (including dzdc at start)
        stopping stop;    // stopping condition
845     N accuracy;       // number of bits to be accurate to (for knighty2)

        // constructor initializes coefficients
        series_approximation(C_hi c, R_lo dt, R_lo tmax, N m, stopping stop, N ↙
            ↳ accuracy)
        : n(1)            // simplifies some things to start at iteration 1 with z = c
850     , dt(dt)
        , tmax(tmax)
        , z(c)
        , c(c)
        , m(m)
855     , a(m + 2)
        , b(m + 1)
        , stop(stop)
        , accuracy(accuracy)
        {
860       assert(dt > 0);
```

```
    assert(tmax > 0);
    assert(m >= 1);
    a[0] = c_lo(c);
    a[1] = C_lo(1);
    for (N k = 2; k <= m + 1; ++k)
      a[k] = C_lo(0); // includes truncation error R at end
    b[0] = a[1];
    for (N k = 1; k <= m; ++k)
      b[k] = C_lo(0);
  };

  // advance if series approximation is valid
  bool step()
  {
    // calculate coefficients
    C_hi z_next(z * z + c);
    vector<C_lo> a_next(m + 2);
    a_next[0] = c_lo(z_next);
    a_next[1] = R_lo(2) * a[0] * a[1] + C_lo(1);
    for (N k = 2; k <= m; ++k)
    {
      C_lo sum(0);
      for (N j = 0; j <= (k - 1) / 2; ++j)
        sum += a[j] * a[k - j];
      sum *= R_lo(2);
      if (! (k & 1)) // if k is even
        sum += a[k/2] * a[k/2];
      a_next[k] = sum;
    }
    // calculate derivative coefficients
    vector<C_lo> b_next(m + 1);
    b_next[0] = a_next[1];
    for (N k = 1; k <= m; ++k)
    {
      C_lo sum(0);
      for (N j = 0; j <= k; ++j)
        sum += a[j] * b[k - j];
      b_next[k] = R_lo(2) * sum;
    }
    // calculate truncation error
    {

/*
      // knighty's first estimate
      vector<R_lo> a_abs(m + 2);
      for (N k = 0; k <= m + 1; ++k)
        a_abs[k] = abs(a[k]);
      R_lo sum(0);
      R_lo tmaxn(1);
      for (N j = 0; j <= m + 1; ++j)
      {
        R_lo sum2(0);
        for (N k = j; k <= m + 1; ++k)
          sum2 += a_abs[k] * a_abs[m + 1 + j - k];
        sum += sum2 * tmaxn;
        tmaxn *= tmax;
      }
```

```
*/

/*
        // knighty's second estimate
        R_lo sum(0);
        R_lo tmaxn(1);
        for (N k = m + 1; k <= 2 * m; ++k)
        {
          C_lo sum2(0);
          for (N j = k - m; j <= (k - 1) / 2; ++j)
            sum2 += a[j] * a[k - j];
          sum2 *= R_lo(2);
          if (! (k & 1)) // if k is even
            sum2 += a[k / 2] * a[k / 2];
          sum += abs(sum2) * tmaxn;
          tmaxn *= tmax;
        }
        tmaxn = R_lo(1);
        R_lo sum2(0);
        for (N j = 0; j <= m; ++j)
        {
          sum2 += abs(a[j]) * tmaxn;
          tmaxn *= tmax;
        }
        sum += 2 * abs(a[m + 1]) * sum2;
        sum += 2 * abs(a[m + 1]) * abs(a[m + 1]) * tmaxn;
        a_next[m + 1] = C_lo(sum);
*/

        // quaz0r's rewrite of knighty's formula
        R_lo sum(0);
        R_lo tmaxk(1);
        for (N k = 0; k <= m - 1; ++k)
        {
          C_lo sum2(0);
          for (N i = k; i <= (m - 1 + k - 1) / 2; ++i)
            sum2 += a[i + 1] * a[m - 1 + k - i + 1];
          sum2 *= R_lo(2);
          if (! (k & 1)) // if k is even
            sum2 += a[k/2 + 1] * a[k/2 + 1];
          sum += tmaxk * abs(sum2);
          tmaxk *= tmax;
        }
        R_lo sum2(abs(a[0]));
        R_lo tmaxi1(tmax);
        for (N i = 0; i <= m - 1; ++i)
        {
          sum2 += abs(a[i + 1]) * tmaxi1;
          tmaxi1 *= tmax;
        }
        sum2 *= R_lo(2) * abs(a[m + 1]);
        sum += sum2;
        sum += abs(a[m + 1]) * abs(a[m + 1]) * tmaxi1;
        a_next[m + 1] = C_lo(sum);

      }
      // check validity of next
```

```
975       bool valid(false);
          switch (stop)
          {

          case knighty:
980         { // knighty's stopping condition
            vector<R_lo> a_abs(m + 2);
            for (N k = 0; k <= m + 1; ++k)
              a_abs[k] = abs(a_next[k]);
            N max_k(1);
985         R_lo max_term(a_abs[1]);
            R_lo tmaxn(1);
            for (N k = 1; k <= m; ++k)
            {
              R_lo term = R_lo(k) * a_abs[k] * tmaxn;
990           if (term > max_term)
              {
                max_term = term;
                max_k = k;
              }
995           tmaxn *= tmax;
            }
            R_lo rhs(max_term);
            tmaxn = 1;
            for (N k = 1; k <= m; ++k)
1000        {
              if (k != max_k)
                rhs -= R_lo(k) * a_abs[k] * tmaxn;
              tmaxn *= tmax;
            }
1005        rhs *= dt;
            R_lo lhs(a_abs[m + 1]);
            for (N k = 0; k < m + 1; ++k)
              lhs *= tmax;
            valid = lhs <= rhs;
1010      }
          break;

          case quaz0r:
            { // quaz0r's stopping condition
1015          vector<R_lo> a_abs(m + 2);
              for (N k = 0; k <= m + 1; ++k)
                a_abs[k] = abs(a_next[k]);
              R_lo zsa(0);
              R_lo tmaxn(1);
1020          for (N k = 0; k <= m; ++k)
              {
                zsa += a_abs[k] * tmaxn;
                tmaxn *= tmax;
              }
1025          R_lo dsa(0);
              tmaxn = 1;
              for (N k = 1; k <= m; ++k)
              {
                dsa += R_lo(k) * a_abs[k] * tmaxn;
1030            tmaxn *= tmax;
              }
```

```
          R_lo rhs(zsa * dsa * dt);
          R_lo lhs(a_abs[m + 1]);
          for (N k = 0; k < m + 1; ++k)
            lhs *= tmax;
          valid = lhs <= rhs;
        }
        break;

      case knighty2:
        { // knighty's second stopping condition
          vector<R_lo> a_abs(m + 2);
          for (N k = 0; k <= m + 1; ++k)
            a_abs[k] = abs(a_next[k]);
          R_lo sa(0);
          R_lo tmaxn(tmax);
          for (N k = 1; k <= m; ++k)
          {
            sa += a_abs[k] * tmaxn;
            tmaxn *= tmax;
          }
          R_lo rhs(sa);
          R_lo lhs(a_abs[m + 1]);
          for (N k = 0; k < m + 1; ++k)
            lhs *= tmax;
          valid = lhs <= R_lo(pow(2.0, -accuracy)) * rhs;
        }
        break;

      default:
        assert(! "valid stopping condition");
        break;
      }
      // advance
      if (valid)
      {
        n = n + 1;
        z = z_next;
        a = a_next;
        b = b_next;
      }
      return valid;
    };

    // keep stepping until invalid
    // TODO FIXME check escape?
    void run()
    {
      LOG(cout << "computing series approximation" << endl);
      while (step())
        if (n % 1000 == 0)
          LOG(cout << "\rseries approximation iteration " << n);
      LOG(cout << "\rseries approximation iteration " << n << " (complete)" << ↙
          ↳ endl);
    };

    // initialize a pixel
    C_lo series_dz(const C_lo &dc1)
```

```
    {
      C_lo dcn(dc1);
      C_lo sum(0);
      for (N k = 1; k <= m; ++k)
      {
        sum += a[k] * dcn;
        dcn *= dc1;
      }
      return sum;
    };

    // initialize a pixel (derivative)
    C_lo series_dzdc(const C_lo &dc1)
    {
      C_lo dcn(1);
      C_lo sum(0);
      for (N k = 0; k <= m; ++k)
      {
        sum += b[k] * dcn;
        dcn *= dc1;
      }
      return sum;
    };

    // get a reference to carry on after series approximation initialisation
    reference get_reference()
    {
      return reference(n, z, c);
    };

  };

  // simple RGB24 image
  class image
  {
  public:
    N width;
    N height;
    vector<uint8_t> rgb;

    // construct empty image
    image(N width, N height)
    : width(width)
    , height(height)
    , rgb(width * height * 3)
    { };

    // plot a point
    void plot(N x, N y, N r, N g, N b)
    {
      N k = (y * width + x) * 3;
      rgb[k++] = r;
      rgb[k++] = g;
      rgb[k++] = b;
    };

    // save to PPM format
```

```
1145    void save(const char *filename)
        {
#ifdef __EMSCRIPTEN__
        (void) filename;
        rgb.push_back(255);
1150    emscripten_worker_respond((char *) &rgb[0], width * height * 3 + 1);
#else
        FILE *out = fopen(filename, "wb");
        fprintf(out, "P6\n%d %d\n255\n", width, height);
        fwrite(&rgb[0], width * height * 3, 1, out);
1155    fflush(out);
        fclose(out);
#endif
        }
};
1160
// wrap around image plot to plot distance estimator colouring
void plot(image &i, const perturbation &p, R_lo dt)
{
  N g(p.n & 255);
1165  if (! p.escaped && ! p.glitched)
    g = 255;
  if (p.escaped)
  {
    R_lo de(R_lo(2) * abs(p.dz) * R_lo(log(to_ld(abs(p.dz)))) / abs(p.dzdc));
1170    g = 255 * tanh(to_ld(de / dt));
  }
  i.plot(p.x, p.y, p.glitched ? 255 : g, g, g);
}

1175 // sort perturbations by n, then within groups by |dz|
bool compare_glitched(const perturbation &p, const perturbation &q)
{
  if (p.n < q.n)
    return true;
1180  if (p.n > q.n)
    return false;
  return norm(p.dz) < norm(q.dz);
}

1185 // solve glitches recursively
N solve_glitches(image &i, R_lo dt, N maxiters, const reference &r0, vector<↙
    ↪ perturbation> &glitches)
{
  N count(0);
  sort(glitches.begin(), glitches.end(), compare_glitched);
1190  // first glitch in groups of equal n has smallest |dz| (set to actual z)
  unsigned int k(0);
  while (k < glitches.size())
  {
    N n(glitches[k].n);
1195    // one iteration of Newton's method with low precision to find next ↙
        ↪ reference
    // gives a few more bits
    C_hi c(r0.c + c_hi(glitches[k].dc) - c_hi(glitches[k].dz / glitches[k].dzdc)↙
        ↪ );
    // rebase delta c against new reference
```

```
      C_lo dc(c_lo(c - r0.c));
      // new reference hits zero at period
      reference r(n, C_hi(0, 0), c); // C_hi needs two args otherwise gives NaN in Boost?
      r.run(maxiters);
      ++count;
      vector<perturbation> subglitches;
      while (k < glitches.size() && glitches[k].n == n)
      {
        // rebase to new reference
        const perturbation &p(glitches[k]);
        perturbation q(p.x, p.y, p.n, p.dc + dc, p.dz, p.dzdc);
        // iterate
        q.run(r, maxiters);
        // output
        if (q.glitched)
          subglitches.push_back(q);
        else
          plot(i, q, dt);
        ++k;
      }
      // recurse
      if (subglitches.size() > 0)
        count += solve_glitches(i, dt, maxiters, r, subglitches);
    }
    return count;
}

void render(N width, N height, N maxiters, N order, N stopn, N accuracy, N precision, const char *sre, const char *sim, const char *sradius, const char *filename)
{
    // prepare
    mpfr_float::default_precision(20u);
    R_lo radius(r_lo(sradius));
    N precision2 = fmax(24.0, 24.0 - double(logb(radius)));
    if (precision && precision < precision2)
      LOG(cout << "warning: insufficient precision: " << precision << " < " << precision2 << endl);
    else if (! precision)
      precision = precision2;
    LOG(cout << "precision bits " << precision << endl);
    precision *= log10(2.0);
    LOG(cout << "precision digits10 " << precision << endl);
    mpfr_float::default_precision(precision);
    C_hi c((R_hi(sre)), (R_hi(sim)));
    // find reference
    C_hi refc(c);
    boxperiod box(c, 2 * radius);
    N period(box.period(maxiters));
    if (! (period > 0))
      period = last_partial(c, maxiters);
    refc = newton_nucleus(c, period, 8); // FIXME best maxsteps value?
    C_lo dc0(c_lo(c - refc));
    LOG(cout << "reference delta " << dc0 << endl);
    // compute dt and tmax
    R_lo dt(radius / (height / 2));
```

```
  R_lo tmax(0);
  for (N y = 0; y < height; y += height - 1)
    for (N x = 0; x < width; x += width - 1)
    {
      C_lo dc(dc0 + R_lo(2) * radius *
        C_lo(width * ((x + 0.5) / width - 0.5) / height, (height - y - 0.5) / ↙
            ↳ height - 0.5));
      R_lo t(abs(dc));
      tmax = max(tmax, t);
    }
  // render
  series_approximation::stopping stop;
  switch (stopn)
  {
    default:
    case 0: stop = series_approximation::knighty;  break;
    case 1: stop = series_approximation::quaz0r;   break;
    case 2: stop = series_approximation::knighty2; break;
  }
  series_approximation s(refc, dt, tmax, order, stop, accuracy);
  s.run();
  reference r(s.get_reference());
  r.run(maxiters);
  image i(width, height);
  N progress = 0;
  vector<perturbation> glitches;
  LOG(cout << "computing image" << endl);
  #pragma omp parallel for schedule(dynamic)
  for (N y = 0; y < height; ++y)
  {
    for (N x = 0; x < width; ++x)
    {
      C_lo dc(dc0 + R_lo(2) * radius *
        C_lo(width * ((x + 0.5) / width - 0.5) / height, (height - y - 0.5) / ↙
            ↳ height - 0.5));
      perturbation p(x, y, r.n0, dc, s.series_dz(dc), s.series_dzdc(dc));
      p.run(r, maxiters);
      plot(i, p, dt);
      if (p.glitched)
      {
        #pragma omp critical
        glitches.push_back(p);
      }
    }
    #pragma omp critical
    LOG(cout << "\rimage scanline " << ++progress);
  }
  LOG(cout << "\rimage scanline " << progress << " (complete)" << endl);
  // correct glitches
  LOG(cout << "glitches " << glitches.size() << endl);
  N count(0);
  if (glitches.size() > 0)
  {
    LOG(cout << "solving glitches" << endl);
    count = solve_glitches(i, dt, maxiters, r, glitches);
  }
  LOG(cout << "secondary references " << count << endl);
```

```
  // save final image
  i.save(filename);
}

#ifdef __EMSCRIPTEN__

// worker thread worker function
extern "C" void EMSCRIPTEN_KEEPALIVE render(char *data, int size)
{
  N width;
  N height;
  N maxiters;
  N order;
  N stop;
  N accuracy;
  N precision;
  const char *sre;
  const char *sim;
  const char *sradius;
  const char *filename;
  unpack(data, size, width, height, maxiters, order, stop, accuracy, precision, ↙
      ↳ sre, sim, sradius, filename);
  render(width, height, maxiters, order, stop, accuracy, precision, sre, sim, ↙
      ↳ sradius, filename);
}

#else

// entry point
extern "C" int main(int argc, char **argv)
{
  // initial defaults
  N width = 640;
  N height = 360;
  N maxiters = 1 << 8;
  N order = 16;
  N stop = 0;
  N accuracy = 24;
  N precision = 0; // auto
  const char *sre = "-0.75";
  const char *sim = "0.0";
  const char *sradius = "1.5";
  const char *filename = "out.ppm";
  // parse arguments
  if (parse(argc, argv, width, height, maxiters, order, stop, accuracy, ↙
      ↳ precision, sre, sim, sradius, filename))
    return 1;
  // render image
  render(width, height, maxiters, order, stop, accuracy, precision, sre, sim, ↙
      ↳ sradius, filename);
  return 0;
}

#endif
```

46 mandelbrot-series-approximation/pre.js

```
function argparse(prev, item)
{
  prev[prev.length] = '--' + item.split('=')[0];
  prev[prev.length] = item.split('=')[1];
  return prev;
}
Module['arguments'] = location.search.substr(1).split('&').reduce(argparse, []);
function printappend(str)
{
  var output = document.getElementById('output');
  var content = output.value;
  var lines = content.split('\n').filter(function(s) { return s != ''; });
  if (str.charAt(0) == '\r')
  {
    lines.pop();
    str = str.substr(1);
  }
  str = str.replace('\n', '');
  lines[lines.length] = str;
  output.value = lines.join('\n');
  output.scrollTop = output.scrollHeight;
}
Module['print'] = printappend;
Module['printErr'] = printappend;
```

47 mandelbrot-winding/Makefile

```
winding: winding.c
	gcc -std=c99 -Wall -Wextra -pedantic -O3 -fopenmp -march=native -o ↙
	    ↳ winding winding.c -lm
```

48 mandelbrot-winding/winding.c

```
// gcc -std=c99 -Wall -Wextra -pedantic -O3 -fopenmp -march=native -o winding ↙
    ↳ winding.c -lm
// ./winding > winding.pgm

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

#define PI 3.141592653589793

static double cnorm(double _Complex z)
{
  return creal(z) * creal(z) + cimag(z) * cimag(z);
}

// based on mndlbrot::turn() from Wolf Jung's Mandel 5.14, under GPL license
// available from http://www.mndynamics.com/indexp.html
static double winding(double _Complex c, int maxiters)
{
  double _Complex z = c;
  double theta = carg(z);
  double s = 1;
```

```
  const double _Complex uv = csqrt(0.25 - c);
  const double _Complex XY = 0.5 - uv;
  const double a = creal(c);
  const double b = cimag(c);
  const double X = creal(XY);
  const double Y = cimag(XY);
  for (int k = 1; k < maxiters; ++k)
  {
    s *= 0.5;
    z = z * z + c;
    double u = carg(z / (z - c));
    const double x = creal(z);
    const double y = cimag(z);
    if ( (y*a - x*b)*(Y*a - X*b) > 0
      && (y*X - x*Y)*(b*X - a*Y) > 0
      && ((b-y)*(a-X) - (a-x)*(b-Y))*(a*Y - b*X) > 0)
    { if (u < 0) u += 2*PI; else u -= 2*PI; }
    theta += s * u;
    if (cnorm(z) > 1e18 * s) break;
    if (cnorm(z) < 1e-12) return 0.5;
  }
  theta *= 0.5 / PI;
  return theta - floor(theta);
}

extern int main(int argc, char **argv)
{
  (void) argc;
  (void) argv;
  double _Complex c0 = -0.75;
  double r0 = 2;
  int maxiters = 64;
  int width = 1024;
  int height = 1024;
  int bytes = width * height;
  char *pgm = malloc(bytes);
  #pragma omp parallel for
  for (int j = 0; j < height; ++j)
  {
    double y = ((j + 0.5) / height - 0.5) * 2;
    for (int i = 0; i < width; ++i)
    {
      double x = ((i + 0.5) / width - 0.5) * 2 * width / height;
      double _Complex c = c0 + r0 * (x + I * y);
      double t = winding(c, maxiters);
      pgm[j * width + i] = 256 * t;
    }
  }
  printf("P5\n%d %d\n255\n", width, height);
  fwrite(pgm, bytes, 1, stdout);
  free(pgm);
  return 0;
}
```

49 pjulia-mdem/Makefile

```
all: pjulia-c pjulia-hs
```

```
clean:
	-rm -f pjulia-c pjulia-hs pjulia.hi pjulia.o pjulia-c.pgm pjulia-hs.pgm

benchmark: pjulia-c pjulia-hs
	bash -c "time ./pjulia-c  -0.390540870218400005 -0.586787907346968729 ↙
	    ↳          > pjulia-c.pgm"
	bash -c "time ./pjulia-hs -0.390540870218400005 -0.586787907346968729 +↙
	    ↳ RTS -N > pjulia-hs.pgm"

pjulia-c: pjulia.c
	gcc -O3 -o pjulia-c -lm -Wall -fopenmp -march=native pjulia.c

pjulia-hs: pjulia.hs
	ghc -O2 -fexcess-precision -Wall -threaded -o pjulia-hs pjulia.hs
```

50 pjulia-mdem/pjulia.c

```
/*

Adam Majewski
fraktal.republika.pl

modified by Claude Heiland-Allen
mathr.co.uk

c console progam using
* symmetry
* openMP

It uses modified DEM method to draw parabolic julia sets


gcc t.c -lm -Wall -fopenmp -march=native
time ./a.out


File s1000000f1.5.pgm saved.
Cx  = 0.356763
Cy  = 0.328582
alfax  = 0.154508
alfay  = 0.475528
distorsion of image  = 1.000000

real    123m19.106s
----------------------------
File s10000000f1.5.pgm saved.
Cx  = 0.356763
Cy  = 0.328582
alfax  = 0.154508
alfay  = 0.475528
distorsion of image  = 1.000000

real    1023m48.596s = 17 fours
-------------------------------------
so s100000000f1.5.pgm shoud take 17 days !!!!!!
```

```
40
45
*/

50 #include <stdio.h>
#include <stdlib.h> // malloc
#include <string.h> // strcat
#include <math.h> // M_PI; needs -lm also
#include <complex.h>
55 #include <omp.h> // OpenMP; needs also -fopenmp

/* ---------------------------------- global variables and consts ↙
    ↳ ------------------------------------------------------------ */

60
// virtual 2D array and integer ( screen) coordinate
// Indexes of array starts from 0 not 1
unsigned int ix, iy; // var
unsigned int ixMin = 0; // Indexes of array starts from 0 not 1
65 unsigned int ixMax ; //
unsigned int iWidth ; // horizontal dimension of array
unsigned int ixAxisOfSymmetry  ; //
unsigned int iyMin = 0; // Indexes of array starts from 0 not 1
unsigned int iyMax ; //
70 unsigned int iyAxisOfSymmetry  ; //
unsigned int iyAbove ; // var, measured from 1 to (iyAboveAxisLength -1)
unsigned int iyAboveMin = 1 ; //
unsigned int iyAboveMax ; //
unsigned int iyAboveAxisLength ; //
75 unsigned int iyBelowAxisLength ; //
unsigned int iHeight = 1000; //  odd number !!!!!! = (iyMax -iyMin + 1) = ↙
    ↳ iyAboveAxisLength + iyBelowAxisLength +1
// The size of array has to be a positive constant integer
unsigned int iSize ; // = iWidth*iHeight;

80
// memmory 1D array
unsigned char *data;
// unsigned int i; // var = index of 1D array
unsigned int iMin = 0; // Indexes of array starts from 0 not 1
85 unsigned int iMax ; // = i2Dsize-1  =
// The size of array has to be a positive constant integer
// unsigned int i1Dsize ; // = i2Dsize  = (iMax -iMin + 1) =  ;  1D array with ↙
    ↳ the same size as 2D array

90 /* world ( double) coordinate = dynamic plane */
  const double ZxMin=-1.5;
  const double ZxMax=1.5;
  const double ZyMin=-1.5;
```

```
  const double ZyMax=1.5;
  double PixelWidth; // =(ZxMax-ZxMin)/iXmax;
  double PixelHeight; // =(ZyMax-ZyMin)/iYmax;
  double distanceMax;
  double ratio ;
  double lambda=1.5;


// complex numbers of parametr plane
double Cx; // c =Cx +Cy * i
double Cy;
double complex c; //

double complex alfa; // alfa fixed point alfa=f(alfa)

unsigned long int iterMax  = 100000; //iHeight*100;
double ER = 2.0; // Escape Radius for bailout test
double ER2;


/* ---------------------------------------- functions ↙
    ↳ ------------------------------------------------------------*/



/* find c in component of Mandelbrot set

 uses code by Wolf Jung from program Mandel
 see function mndlbrot::bifurcate from mandelbrot.cpp
 http://www.mndynamics.com/indexp.html

  */
static
double complex GiveC(double InternalAngleInTurns, double InternalRadius, ↙
    ↳ unsigned int period)
{
  //0 <= InternalRay<= 1
  //0 <= InternalAngleInTurns <=1
  double t = InternalAngleInTurns *2*M_PI; // from turns to radians
  double R2 = InternalRadius * InternalRadius;
  //double Cx, Cy; /* C = Cx+Cy*i */
  switch ( period ) // of component
  {
    case 1: // main cardioid
      Cx = (cos(t)*InternalRadius)/2-(cos(2*t)*R2)/4;
      Cy = (sin(t)*InternalRadius)/2-(sin(2*t)*R2)/4;
      break;
   case 2: // only one component
      Cx = InternalRadius * 0.25*cos(t) - 1.0;
      Cy = InternalRadius * 0.25*sin(t);
      break;
  // for each period  there are 2^(period-1) roots.
  default: // higher periods : to do
      Cx = 0.0;
      Cy = 0.0;
    break; }
```

```
150     return Cx + Cy*I;
    }


    /*
155
    http://en.wikipedia.org/wiki/Periodic_points_of_complex_quadratic_mappings
    z^2 + c = z
    z^2 - z + c = 0
    ax^2 +bx + c =0 // ge3neral for  of quadratic equation
160 so :
    a=1
    b =-1
    c = c
    so :
165
    The discriminant is the  d=b^2- 4ac

    d=1-4c = dx+dy*i
    r(d)=sqrt(dx^2 + dy^2)
170 sqrt(d) = sqrt((r+dx)/2)+-sqrt((r-dx)/2)*i = sx +- sy*i

    x1=(1+sqrt(d))/2 = beta = (1+sx+sy*i)/2

    x2=(1-sqrt(d))/2 = alfa = (1-sx -sy*i)/2
175
    alfa : attracting when c is in main cardioid of Mandelbrot set, then it is in ↙
        ↳ interior of Filled-in Julia set,
    it means belongs to Fatou set ( strictly to basin of attraction of finite fixed ↙
        ↳ point )

    */
180 // uses global variables :
    //  ax, ay (output = alfa(c))
    static
    double complex GiveAlfaFixedPoint(double complex c)
    {
185   double dx, dy; //The discriminant is the  d=b^2- 4ac = dx+dy*i
      double r; // r(d)=sqrt(dx^2 + dy^2)
      double sx, sy; // s = sqrt(d) = sqrt((r+dx)/2)+-sqrt((r-dx)/2)*i = sx + sy*i
      double ax, ay;

190   // d=1-4c = dx+dy*i
      dx = 1 - 4*creal(c);
      dy = -4 * fabs(cimag(c));
      // r(d)=sqrt(dx^2 + dy^2)
      r = sqrt(dx*dx + dy*dy);
195   //sqrt(d) = s =sx +sy*i
      sx = sqrt((r+dx)/2);
      sy = sqrt((r-dx)/2);
      // alfa = ax +ay*i = (1-sqrt(d))/2 = (1-sx + sy*i)/2
      ax = 0.5 - sx/2.0;
200   ay = (cimag(c) > 0 ? 1 : -1) * sy/2.0;


    return ax+ay*I;
```

```
}

static
double GiveDistance2Between(double complex z1, double complex z2 )
{double dx,dy;

 dx = creal(z1) - creal(z2);
 dy = cimag(z1) - cimag(z2);
 return (dx*dx+dy*dy);

}



static
int setup(int argc, char **argv)
{
  if (argc >= 3) {
    Cx = atof(argv[1]);
    Cy = atof(argv[2]);
    c = Cx + Cy * I;
  } else {
    double InternalAngle;

    InternalAngle = (sqrt(5) - 1)/2;

    c = GiveC(InternalAngle, 1.0, 1) ;
    Cx=creal(c);
    Cy=cimag(c);
  }
  alfa = GiveAlfaFixedPoint(c);

  /* 2D array ranges */
  if (!(iHeight % 2)) iHeight+=1; // it sholud be even number (variable % 2) or ↙
      ↳ (variable & 1)
  iWidth = iHeight;
  iSize = iWidth*iHeight; // size = number of points in array
  // iy
  iyMax = iHeight - 1 ; // Indexes of array starts from 0 not 1 so the highest ↙
      ↳ elements of an array is = array_name[size-1].
  iyAboveAxisLength = (iHeight -1)/2;
  iyAboveMax = iyAboveAxisLength ;
  iyBelowAxisLength = iyAboveAxisLength; // the same
  iyAxisOfSymmetry = iyMin + iyBelowAxisLength ;
  // ix

  ixMax = iWidth - 1;

  /* 1D array ranges */
  // i1Dsize = i2Dsize; // 1D array with the same size as 2D array
  iMax = iSize-1; // Indexes of array starts from 0 not 1 so the highest ↙
      ↳ elements of an array is = array_name[size-1].


 /* Pixel sizes */
  PixelWidth = (ZxMax-ZxMin)/ixMax; //  ixMax = (iWidth-1)  step between pixels ↙
```

```
    ↪ in world coordinate
  PixelHeight = (ZyMax-ZyMin)/iyMax;
  ratio = ((ZxMax-ZxMin)/(ZyMax-ZyMin))/((float)iWidth/(float)iHeight); // it ↙
    ↪ should be 1.000 ...
  distanceMax = PixelWidth;
  //

  ER2 = ER * ER;

  /* create dynamic 1D arrays for colors ( shades of gray ) */

  data = malloc( iSize * sizeof(unsigned char) );
  if (data == NULL )
    {
      fprintf(stderr," Could not allocate memory\n");
      getchar();
      return 1;
    }
  else fprintf(stderr," memory is OK \n");


  return 0;

}



// from screen to world coordinate ; linear mapping
// uses global cons
static
double GiveZx(unsigned int ix)
{ return (ZxMin + ix*PixelWidth );}

// uses globaal cons
static
double GiveZy(unsigned int iy)
{ return (ZyMax - iy*PixelHeight);} // reverse y axis


/*
 estimates distance from point c to nearest point in Julia  set
 for Fc(z)= z*z + c
 z(n+1) = Fc(zn)
 this function is based on function  mndlbrot::dist  from  mndlbrot.cpp
 from program mandel by Wolf Jung (GNU GPL )
 http://www.mndynamics.com/indexp.html

Hyunsuk Kim  :
For Julia sets, z is the variable and c is a constant. Therefore df[n+1](z)/dz =↙
    ↪  2*f[n]*f'[n] -- you don't add 1.
-----------------


"
The algorithm I am using is a mix of techinques I have been taught by other ↙
    ↪ people.
```

```
This picture was drawn by scanning all the pixels and computing the color of ↙
    ↳ each of them by iterating the corresponding complex number and seeing what↙
    ↳  happens.

One of the technique is very general in its applications. Let P be the function ↙
    ↳ which you want to computer the Julia set, when you inductively compute
z_n+1=P(z_n)=P^n(z_0),
also compute the derivative
d_n=(P^n)'(z_0)
It can be done inductively by the chain rule: d_n+1=P'(z_n).d_n

Then the idea is to pretend that the image of the pixel of size epsilon by P^n ↙
    ↳ is roughly of size epsilon.d_n. This is not always accurate but works ↙
    ↳ surprisingly well in the pictures.

Then the trick is to choose a point 'p' that we know to belong to the Julia set.↙
    ↳  Now when you iterate, test wether the distance from z_n to 'p' is less ↙
    ↳ than epsilon.d_n If this is so, then there is a good chance that z_0 is ↙
    ↳ within distance of order epsilon to the Julia set, so color your point in ↙
    ↳ black (or whichever color you chose for the Julia set).

Usually, 'p'= a repelling fixed point of P is a good choice.

For Siegel disks and Herman rings, a good choice of 'p' is the critical point 'c↙
    ↳ '.

Namely,
if |z_n-c|<epsilon*d_n
then colored z_0 in black.

Your Herman ring can be enhanced a lot using the tricks which described above."

Regards,
Yang Fei

A pixel of size epsilon, after iterating n times by P,
we obtain a domain is roughly of size epsilon*d_n, where d_n means the ↙
    ↳ derivative of P^n at z_0.


------------------------------
"The following method was taught to me by Christian Henriksen.
It is a distance estimator method (different from Milnor's).
When iterating forward, keep also tract of the value of the derivative.
More precisely when computing z_n from z_0, also compute dz_n/dz_0.
by the chain rule, it is equal to the product of the derivatives of f taken at ↙
    ↳ z_0, z_1, .. , up to z_n-1
Therefore you can compute it recursively alongside and like the value of z_n.
Then choose a point in the Julia set, here I chose the critical point.
Then if the ball :
- of center z_n
- and radius lambda * size of a pixel * derivative
contains this point there is a good chance
that the pixel of center z_0
contains an iterated preimage of the critical point, thus I color z_0 in black.
```

```
Here lambda is a thickness factor that you must choose.
not too small otherwise you do not see enough points, but not too big because ↙
    ↳ otherwise you get artifacts.

Best Regards,
Arnaud."


 */
static
 int jdist(double Zx, double Zy, double Cx, double Cy ,  int iter_max)
 {
 int i;
 double x = Zx, /* Z = x+y*i */
        y = Zy,
        /* Zp = xp+yp*1 = 1  */
        xp = 1,
        yp = 0,
        /* temporary */
        nz,
        nzp,
        /* a = abs(z) */
        //a,
        radius2,
        distance2;



 for (i = 1; i <= iter_max; i++)
  { /* first derivative   zp = 2*z*zp  = xp + yp*i; */

    nz = 2*(x*xp - y*yp) ;
    yp = 2*(x*yp + y*xp);
    xp = nz;
    /* z = z*z + c = x+y*i */
    nz = x*x - y*y + Cx;
    y = 2*x*y + Cy;
    x = nz;
    //
    nz = (x*x + y*y); // abs2(z)
    nzp = (xp*xp + yp*yp);
    if ( nz > ER2) return 1; // if escapes //nzp > 1e60 ||
    if (nzp>1e60) return -1; // ? interior ?
    distance2=GiveDistance2Between(alfa,x+y*I);
    radius2=lambda*lambda*PixelWidth*PixelWidth*nzp;

    if (distance2<radius2)return 0; //
  }

 return -1; //
 }


static
unsigned char GiveColor(unsigned int ix, unsigned int iy)
{
   double Zx, Zy; //  Z= Zx+ZY*i;
```

```
  unsigned char color; // gray from 0 to 255
// unsigned char ColorList[]={255,230,180}; /* shades of gray used in imaage
  //color=0;
  // from screen to world coordinate
  Zx = GiveZx(ix);
  Zy = GiveZy(iy);
  //if ( GiveLastIteration(Zx, Zy ) == iterMax)
    // color = 0; // interior
    // else {

// only dist
  int d = jdist(Zx, Zy, Cx, Cy , iterMax);
       if (d > 0) { color = 255; } // exterior Fatou Set
  else if (d < 0) { color = 192; } // interior Fatou Set
  else            { color =   0; } // boundary Julia set
  return color;
}


/* -----------  array functions -------------- */


/* gives position of 2D point (iX,iY) in 1D array  ; uses also global variable ↙
    ↳ iWidth */
static
unsigned int Give_i(unsigned int ix, unsigned int iy)
{ return ix + iy*iWidth; }
//  ix = i % iWidth;
//  iy = (i- ix) / iWidth;
//  i  = Give_i(ix, iy);




// plots raster point (ix,iy)
static
int PlotPoint(unsigned int ix, unsigned int iy, unsigned char iColor)
{
 unsigned i; /* index of 1D array */
 i = Give_i(ix,iy); /* compute index of 1D array from indices of 2D array */
 data[i] = iColor;

return 0;
}


// fill array using symmetry of image
// uses global var :  ...
static
int FillArraySymmetric(unsigned char data[] )
{

 unsigned char Color; // gray from 0 to 255

fprintf(stderr, "axis of symmetry \n");
iy = iyAxisOfSymmetry;
#pragma omp parallel for schedule(dynamic) private(ix,Color) shared(ixMin,ixMax,↙
```

```
    ↪ iyAxisOfSymmetry)
for(ix=ixMin;ix<=ixMax;++ix) {//printf(" %d from %d\n", ix, ixMax); //info
                              PlotPoint(ix, iy, GiveColor(ix, iy));
}


/*
The use of 'shared(variable, variable2) specifies that these variables should be↙
    ↪ shared among all the threads.
The use of 'private(variable, variable2)' specifies that these variables should ↙
    ↪ have a seperate instance in each thread.
*/

#pragma omp parallel for schedule(dynamic) private(iyAbove,ix,iy,Color) shared(↙
    ↪ iyAboveMin, iyAboveMax,ixMin,ixMax, iyAxisOfSymmetry)

// above and below axis
for(iyAbove = iyAboveMin; iyAbove<=iyAboveMax; ++iyAbove)
  {fprintf(stderr, " %4d from %d\r", iyAbove, iyAboveMax);fflush(stdout); //info
  for(ix=ixMin; ix<=ixMax; ++ix)

  { // above axis compute color and save it to the array
    iy = iyAxisOfSymmetry + iyAbove;
    Color = GiveColor(ix, iy);
    PlotPoint(ix, iy, Color );
    // below the axis only copy Color the same as above without computing it
    PlotPoint(ixMax-ix, iyAxisOfSymmetry - iyAbove , Color );
   }
}
 return 0;
}


// save data array to pgm file
static
int SaveArray2PGMFile( unsigned char data[])
{

  const unsigned int MaxColorComponentValue=255; /* color component is coded ↙
      ↪ from 0 to 255 ;  it is 8 bit color file */
  char *comment="# ";/* comment should start with # */

  /* save image to the pgm file  */
  fprintf(stdout,"P5\n %s\n %u %u\n %u\n",comment,iWidth,iHeight,↙
      ↪ MaxColorComponentValue);  /*write header to the file*/
  fflush(stdout);
  fwrite(data,iSize,1,stdout);  /*write image data bytes to the file in one step↙
      ↪ */
  return 0;
}


static
int info()
{
  // diplay info messages
  fprintf(stderr, "Cx  = %.18f \n", Cx);
```

```
  fprintf(stderr, "Cy  = %.18f \n", Cy);
  fprintf(stderr, "alfax  = %.18f \n", creal(alfa));
  fprintf(stderr, "alfay  = %.18f \n", cimag(alfa));
  fprintf(stderr, "distorsion of image  = %.18f \n", ratio);
return 0;
}


/* ---------------------------------------  main  ↙
    ↳ ------------------------------------------------------------*/
extern
int main(int argc, char **argv)
{
// here are procedures for creating image file

  setup(argc, argv);
  // compute colors of pixels = image
  //FillArray( data ); // no symmetry
  FillArraySymmetric(data);
  //  CheckOrientation( data );
  SaveArray2PGMFile( data); // save array (image) to pgm file
  free(data);
  info();
  return 0;
}
```

51 pjulia-mdem/pjulia.hs

```
{-
Claude Heiland-Allen
mathr.co.uk

translated from C code by

Adam Majewski
fraktal.republika.pl

-}

{-# LANGUAGE BangPatterns, MagicHash #-}
module Main (main) where

import GHC.Prim ((-#), (==#), (<##), (-##), (+##), (*##))
import GHC.Types (Int(..), Double(..))
import GHC.Conc (getNumCapabilities)
import Control.Parallel.Strategies (parBuffer, rseq, using)
import qualified Data.ByteString as BSS
import qualified Data.ByteString.Lazy as BS
import qualified Data.ByteString.Lazy.Char8 as BSC
import Data.Complex (Complex((:+)), magnitude)
import Data.Word (Word8)
import System.IO (stdout, stderr, hPutStrLn)
import System.Environment (getArgs)
import System.Exit (exitFailure)

type B = Word8
type N = Int
```

```
type R = Double
type C = Complex R

width :: N
width  = 500

height :: N
height = 500

maxIters :: N
maxIters = 100000

zSize :: C
zSize = 1.5

lambda :: R
lambda = 1.5

pixelSize :: R
pixelSize = magnitude (zSize / fromIntegral width)

lambdaPixel2 :: R
lambdaPixel2 = lambda * lambda * pixelSize * pixelSize

er :: R
er = 2

er2 :: R
er2 = er * er

alfa :: C -> C
alfa (cx :+ cy) = ax :+ ay
  where
    dx = 1 - 4 * cx
    dy = -4 * abs cy
    d = dx :+ dy
    r = magnitude d
    sx = sqrt ((r + dx) / 2)
    sy = sqrt ((r - dx) / 2)
    ax = 0.5 - sx / 2
    ay = signum cy * sy / 2

coord :: N -> N -> C
coord j i =
  zSize *   (fromIntegral i / fromIntegral width
  :+ negate (fromIntegral j / fromIntegral height))

data Pixel = Interior | Boundary | Exterior

colour :: Pixel -> B
colour Interior = 192
colour Boundary = 0
colour Exterior = 255

pixel :: C -> C -> C -> Pixel
pixel (D# ax :+ D# ay) (D# cx :+ D# cy) (D# x :+ D# y) = go maxIters' 1.0## ↙
    ↳ 0.0## x y
```

```
  where
    !(I# maxIters') = maxIters
    !(D# lambdaPixel2') = lambdaPixel2
    !(D# er2') = er2
    go !n !dzx !dzy !zx !zy
      | n      ==# 0#  = Interior
      | er2'   <## z2  = Exterior
      | 1e60## <## dz2 = Interior
      | d2     <## r2  = Boundary
      | otherwise  = go (n -# 1#) dzx' dzy' zx' zy'
      where
        dzx' = 2.0## *## ((zx *## dzx) -## (zy *## dzy))
        dzy' = 2.0## *## ((zx *## dzy) +## (zy *## dzx))
        zx'  = (zx2 -## zy2) +## cx
        zy'  = (2.0## *## (zx *## zy)) +## cy
        zx2 = zx *## zx
        zy2 = zy *## zy
        z2  = zx2 +## zy2
        dz2 = (dzx *## dzx) +## (dzy *## dzy)
        azx = ax -## zx
        azy = ay -## zy
        d2  = (azx *## azx) +## (azy *## azy)
        r2  = lambdaPixel2' *## dz2

raster :: N -> C -> BS.ByteString
raster cores c = bulk `BS.append` axis `BS.append` BS.reverse bulk
  where
    bs = ((-height, -width), (-1, width))
    cs = ((0, -width), (0, width))
    bulk = render bs
    axis = render cs
    a = alfa c
    go = colour . pixel a c . uncurry coord
    render = BS.fromChunks . mapP cores (BSS.pack . map go) . ranges

ranges :: ((N, N), (N, N)) -> [[(N, N)]]
ranges ((ly, lx), (hy, hx)) = [[(y, x) | x <- [y + lx - y .. hx]] | y <- [ly .. ↙
    ↪ hy]]

mapP :: N -> (a -> b) -> [a] -> [b]
mapP cores f xs = map f xs `using` parBuffer cores rseq

main' :: N -> C -> BS.ByteString
main' cores c = pgm `BS.append` raster cores c

pgm :: BS.ByteString
pgm = BSC.pack $ "P5\n" ++ show (2 * width + 1) ++ " " ++ show (2 * height + 1) ↙
    ↪ ++ "\n255\n"

main :: IO ()
main = do
  cores <- getNumCapabilities
  args <- getArgs
  case args of
    [sx, sy] -> BS.hPut stdout (main' cores (read sx :+ read sy))
    _ -> hPutStrLn stderr "usage: ./pjulia cx cy > out.pgm" >> exitFailure
```

52 README

```
pjulia-mdem - modified distance estimator method for parabolic Julia sets
mandelbrot-delta-cl - Mandelbrot set perturbation renderer using OpenCL
mandelbrot-series-approximation - Mandelbrot set series approximation example
```

53 trustworthy-anti-buddhagram/.gitignore

```
trustworthy-anti-buddhagram
hyper-voxel-viewer
*.raw
*.mp4
*.jpg
```

54 trustworthy-anti-buddhagram/hyper-voxel-viewer.cpp

```
#define _GNU_SOURCE
#define _FILE_OFFSET_BITS 64
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>
#ifndef MAP_HUGE_2MB
#define MAP_HUGE_2MB (21 << MAP_HUGE_SHIFT)
#endif
#ifndef MAP_HUGE_1GB
#define MAP_HUGE_1GB (30 << MAP_HUGE_SHIFT)
#endif

#include <cstdlib>
#include <ctime>
#include <atomic>
#include <iostream>
#include <sstream>
#include <fstream>
#include <iterator>
#include <vector>

#include <glm/glm.hpp>


float srgb(float c)
{
  c = glm::clamp(c, 0.0f, 1.0f);
  const float a = 0.055;
  if (c <= 0.0031308)
    return 12.92 * c;
  else
    return (1.0 + a) * std::pow(c, 1.0 / 2.4) - a;
}

glm::vec3 srgb(const glm::vec3 &c)
{
  return glm::vec3(srgb(c.x), srgb(c.y), srgb(c.z));
}

uint32_t hash(uint32_t a)
```

```
{
  a = (a+0x7ed55d16u) + (a<<12u);
  a = (a^0xc761c23cu) ^ (a>>19u);
  a = (a+0x165667b1u) + (a<<5u);
  a = (a+0xd3a2646cu) ^ (a<<9u);
  a = (a+0xfd7046c5u) + (a<<3u);
  a = (a^0xb55a4f09u) ^ (a>>16u);
  return a;
}

float uniform(uint32_t h)
{
  return h / (float) UINT_MAX * 2 - 1;
}

unsigned char dither8(float c, uint32_t h)
{
  return std::floor(glm::clamp(float(255 * c + h / (float) UINT_MAX), 0.f, 255.f↙
      ↳ ));
}

glm::vec4 unit_quaternion(uint32_t seed)
{
  glm::vec4 q;
  uint32_t k = 0;
  do
  {
    q = glm::vec4(uniform(hash((k + 0) ^ seed)), uniform(hash((k + 1) ^ seed)), ↙
        ↳ uniform(hash((k + 2) ^ seed)), uniform(hash((k + 3) ^ seed)));
    k += 4;
  } while (glm::length(q) > 1);
  return glm::normalize(q);
}

glm::mat4 quaternion_matrix_l(const glm::vec4 &q)
{
  return glm::mat4
    ( q.x, -q.y, -q.z, -q.w
    , q.y,  q.x, -q.w,  q.z
    , q.z,  q.w,  q.x, -q.y
    , q.w, -q.z,  q.y,  q.x
    );
}

glm::mat4 quaternion_matrix_r(const glm::vec4 &q)
{
  return glm::mat4
    ( q.x, -q.y, -q.z, -q.w
    , q.y,  q.x,  q.w, -q.z
    , q.z, -q.w,  q.x,  q.y
    , q.w,  q.z, -q.y,  q.x
    );
}

size_t z_order(size_t x, size_t y, size_t z, size_t w)
{
  size_t o = 0;
```

```
  for (size_t i = 0; i < sizeof(x) * CHAR_BIT / 4; i++)
  {
    o |= (x & 1LLU << i) << (3 * i + 0)
      |  (y & 1LLU << i) << (3 * i + 1)
      |  (z & 1LLU << i) << (3 * i + 2)
      |  (w & 1LLU << i) << (3 * i + 3)
      ;
  }
  return o;
}

int main(int argc, char **argv)
{
  if (! (argc > 1))
  {
    std::fprintf(stderr, "usage: %s level\n", argv[0]);
    return 1;
  }
  uint32_t seed0 = std::time(0);
  std::fprintf(stderr, "seed %08x\n", seed0);
  size_t n = std::atoll(argv[1]);
  size_t bytes = 2LLU << (4 * n - 3);
  std::ostringstream blackfile;
  blackfile << n << "/black.raw";
  int fd_black = open(blackfile.str().c_str(), O_RDONLY | O_NOATIME);
  if (fd_black < 0)
  {
    fprintf(stderr, "could not open %s\n", blackfile.str().c_str());
    return 1;
  }
  std::ostringstream grayfile;
  grayfile << n << "/gray.raw";
  int fd_gray = open(grayfile.str().c_str(), O_RDONLY | O_NOATIME);
  if (fd_gray < 0)
  {
    fprintf(stderr, "could not open %s\n", grayfile.str().c_str());
    return 1;
  }
  unsigned char *voxels;
  voxels = (unsigned char *) mmap(nullptr, bytes, PROT_READ | PROT_WRITE, ↙
    ↳ MAP_PRIVATE | MAP_HUGETLB | MAP_HUGE_1GB | MAP_ANONYMOUS, -1, 0);
  if (voxels == MAP_FAILED)
  {
    fprintf(stderr, "could not mmap with 1GB pages\n");
    voxels = (unsigned char *) mmap(nullptr, bytes, PROT_READ | PROT_WRITE, ↙
      ↳ MAP_PRIVATE | MAP_HUGETLB | MAP_HUGE_2MB | MAP_ANONYMOUS, -1, 0);
    if (voxels == MAP_FAILED)
    {
      fprintf(stderr, "could not mmap with 2MB pages\n");
      voxels = (unsigned char *) mmap(nullptr, bytes, PROT_READ | PROT_WRITE, ↙
        ↳ MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
      if (voxels == MAP_FAILED)
      {
        fprintf(stderr, "could not mmap at all\n");
        return 1;
      }
      else
```

```
150         {
              fprintf(stderr, "did mmap with 4kB pages\n");
            }
          }
          else
155       {
            fprintf(stderr, "did mmap with 2MB pages\n");
          }
        }
        else
160     {
          fprintf(stderr, "did mmap with 1GB pages\n");
        }
        int fd[2] = { fd_black, fd_gray };
        for (int file = 0; file < 2; ++file)
165     {
          size_t read_bytes = 0;
          do
          {
            /*
170          On Linux, read() (and similar system calls) will transfer at most
             0x7ffff000 (2,147,479,552) bytes, returning the number of bytes actu-
             ally transferred. (This is true on both 32-bit and 64-bit systems.)
            */
            ssize_t result = read(fd[file], voxels + (bytes >> 1) * file + read_bytes,↙
                ↪ (bytes >> 1) - read_bytes);
175         if (result <= 0)
            {
              break;
            }
            read_bytes += result;
180       } while (read_bytes < (bytes >> 1));
          close(fd[file]);
          if (read_bytes != (bytes >> 1))
          {
            fprintf(stderr, "did not read voxel data %ld != %lu\n", read_bytes, bytes)↙
                ↪ ;
185         return 1;
          }
        }
        const glm::vec4 eye(0, 0, 0, -8);
        const glm::mat4 ql(quaternion_matrix_l(unit_quaternion(hash(-1 ^ seed0))));
190     const glm::mat4 qr(quaternion_matrix_r(unit_quaternion(hash(-2 ^ seed0))));
        const float xmin = -2, xmax = 2;
        const float ymin = -2, ymax = 2;
        const float zmin = -2, zmax = 2;
        const float wmin = -2, wmax = 2;
195     const size_t x_tiles = 16, y_tiles = 9;
        const size_t x_steps = 120*2, y_steps = 120*2;
        const size_t z_steps = x_tiles * y_tiles;
        const size_t w_steps = 1 << n;
        const size_t image_width  = x_tiles * x_steps;
200     const size_t image_height = y_tiles * y_steps;
        const size_t subframes = 1;
        const float f50 = std::pow(0.01f, 1.0f / w_steps);
        const float f75 = std::pow(0.1f,  1.0f / w_steps);
        glm::vec4 colours[4] =
```

```
    { glm::vec4(f50, f75, 1.f, 1.f) // "black"  -> blue
    , glm::vec4(1.f, f75, f50, 1.f) // "gray"   -> orange
    , glm::vec4(1.f, 1.f, 1.f, 1.f) // "white"  -> white
    , glm::vec4(1.f, 0.f, 0.f, 1.f) // "marked" -> red
    };
  size_t m = 1LLU << n;
  std::vector<unsigned char> raster;
  raster.reserve(image_width * image_height * 3);
  for (size_t frame = 0; frame < 360; frame += 3)
  {
    for (size_t tile_y = 0; tile_y < y_tiles; ++tile_y)
    {
      #pragma omp parallel for
      for (size_t sub_y = 0; sub_y < y_steps; ++sub_y)
      {
        for (size_t tile_x = 0; tile_x < x_tiles; ++tile_x)
        {
          for (size_t sub_x = 0; sub_x < x_steps; ++sub_x)
          {
            size_t sub_z = tile_y * x_tiles + tile_x;
            glm::vec4 accumulated(0, 0, 0, 0);
            uint32_t seed1 =
                hash(sub_x ^
                hash(tile_x ^
                hash(sub_y ^
                hash(tile_y ^
                hash(frame ^
                hash(seed0))))));
            for (size_t subframe = 0; subframe < subframes; ++subframe)
            {
              uint32_t seed = hash(subframe ^ seed1);
              float t = glm::radians(float(frame + uniform(hash(1 ^ seed))));
              float co = std::cos(t);
              float si = std::sin(t);
              glm::mat4 rot
                ( 1.0f, 0.0f, 0.0f, 0.0f
                , 0.0f, 1.0f, 0.0f, 0.0f
                , 0.0f, 0.0f,  co, si
                , 0.0f, 0.0f, -si, co
                );
              float x = glm::mix(xmin, xmax, (sub_x + uniform(hash(2 ^ seed))) /
                  x_steps);
              float y = glm::mix(ymin, ymax, (sub_y + uniform(hash(3 ^ seed))) /
                  y_steps);
              float z = glm::mix(zmin, zmax, (sub_z + uniform(hash(4 ^ seed))) /
                  z_steps);
              glm::vec4 target(x, y, z, 0);
              glm::vec4 dir = glm::normalize(target - eye);
              // (eye + begin * dir).w = w
              float begin = (wmin - eye.w) / dir.w;
              float end   = (wmax - eye.w) / dir.w;
              glm::vec4 colour(1, 1, 1, 1);
              for (size_t sub_w = 0; sub_w < w_steps; ++sub_w)
              {
                float d = glm::mix(begin, end, (sub_w + uniform(hash((5 + sub_w)
                    ^ seed))) / w_steps);
                glm::vec4 q(eye + d * dir);
```

```
                glm::vec4 p(rot * ql * q * qr);
                glm::vec4 r(float(m) * (p / 2.0f + glm::vec4(1.f, 1.f, 1.f, 1.f)↙
                    ↳ ) / 2.0f);
260             if ( 0 <= r.x && r.x < m &&
                     0 <= r.y && r.y < m &&
                     0 <= r.z && r.z < m &&
                     0 <= r.w && r.w < m )
                {
265               const size_t i = r.x;
                  const size_t j = r.y;
                  const size_t k = r.z;
                  const size_t l = r.w;
                  const size_t ix = z_order(i, j, k, l);
270               const size_t byte = ix >> 3;
                  const size_t bit = ix & 7;
                  const unsigned char mask = 1u << bit;
                  const bool is_black = !!(voxels[byte                ] & mask);
                  const bool is_gray  = !!(voxels[byte + (bytes >> 1)] & mask);
275               const int c = is_black ? 0 : is_gray ? 1 : 2;
                  colour *= colours[c];
                }
              }
              accumulated += colour;
280         }
            glm::vec3 rgb = srgb(glm::vec3(accumulated) / accumulated.w);
            size_t i = tile_x * x_steps + sub_x;
            size_t j = tile_y * y_steps + sub_y;
            size_t k = 3 * (image_width * j + i);
285         raster[k++] = dither8(rgb.x, hash((subframes + 0) ^ seed1));
            raster[k++] = dither8(rgb.y, hash((subframes + 1) ^ seed1));
            raster[k++] = dither8(rgb.z, hash((subframes + 2) ^ seed1));
          }
        }
290   }
    }
    std::fprintf(stdout, "P6\n%lu %lu\n255\n", image_width, image_height);
    std::fwrite(&raster[0], image_width * image_height * 3, 1, stdout);
    std::fflush(stdout);
295 }
  munmap(voxels, bytes);
  return 0;
}
```

55 trustworthy-anti-buddhagram/Makefile

```
all: hyper-voxel-viewer trustworthy-anti-buddhagram

hyper-voxel-viewer: hyper-voxel-viewer.cpp
	g++ -std=c++14 -Wall -Wextra -pedantic -O3 -march=native -fopenmp -g -o ↙
	    ↳ hyper-voxel-viewer hyper-voxel-viewer.cpp
5
trustworthy-anti-buddhagram: trustworthy-anti-buddhagram.cpp
	g++ -std=c++14 -Wall -Wextra -pedantic -O3 -march=native -fopenmp -g -o ↙
	    ↳ trustworthy-anti-buddhagram trustworthy-anti-buddhagram.cpp
```

56 trustworthy-anti-buddhagram/trustworthy-anti-buddhagram.cpp

```cpp
#include <algorithm>
#include <cmath>
#include <cstdio>
#include <fstream>
#include <iostream>
#include <sstream>
#include <utility>
#include <vector>

typedef float real;
real sqr(const real &x)
{
  return x * x;
}

template <typename R>
class complex
{
  R x, y;
public:
  complex(const R &x, const R &y)
  : x(x), y(y)
  { }
  const R &real() const
  {
    return x;
  }
  const R &imag() const
  {
    return y;
  }
};

template <typename R>
complex<R> sqr(const complex<R> &z)
{
  return complex<R>(sqr(z.real()) - sqr(z.imag()), 2 * z.real() * z.imag());
}

template <typename R>
complex<R> operator+(const complex<R> &a, const complex<R> &b)
{
  return complex<R>(a.real() + b.real(), a.imag() + b.imag());
}

template <typename R>
complex<R> operator-(const complex<R> &a, const complex<R> &b)
{
  return complex<R>(a.real() - b.real(), a.imag() - b.imag());
}

enum color
{
  black  = 0,
  gray   = 1,
  white  = 2,
  marked = 3
```

```
};

const color unmarked[4] = { black, black, white, gray };
const color inherit[4] = { black, gray, white, marked };
const unsigned char colorrgb[4][3] = { { 0, 0, 0 }, { 128, 128, 128 }, { 255, ↙
    ↳ 255, 255 }, { 255, 0, 0 } };

class index
{
public:
  size_t x, y;
  index(size_t x, size_t y)
  : x(x), y(y)
  {
  }
  bool operator<(const index &that) const
  {
    return
      x != that.x
    ? x <  that.x
    : y != that.y
    ? y <  that.y
    : false; // equal
  }
  bool operator!=(const index &that) const
  {
    return *this < that || that < *this;
  }
  size_t operator()(size_t n) const
  {
    return (x << n) + y;
  }
};

const real infinity = 1.0 / 0.0;

class range
{
  real r[2];
public:
  range() // empty
  {
    r[0] = infinity;
    r[1] = -infinity;
  }
  range(const real &a)
  {
    r[0] = a;
    r[1] = a;
  }
  range(const real &a, const real &b)
  {
    r[0] = a;
    r[1] = b;
  }
  const real &min() const
  {
```

```
    return r[0];
  }
  const real &max() const
  {
    return r[1];
  }
  bool operator<(const range &that) const
  {
    return
      min() != that.min()
    ? min() <  that.min()
    : max() != that.max()
    ? max() <  that.max()
    : false;
  }
  bool operator!=(const range &that) const
  {
    return *this < that || that < *this;
  }
};

bool contains(const range &a, const real &x)
{
  return a.min() <= x && x <= a.max();
}

real midpoint(const range &a)
{
  return (a.max() + a.min()) / 2;
}

real radius(const range &a)
{
  return (a.max() - a.min()) / 2;
}

range midpoint_radius(const real &a, const real &r)
{
  return range(a - r, a + r);
}

range abs(const range &a)
{
  auto x = std::abs(a.min());
  auto y = std::abs(a.max());
  return range(std::min(x, y), std::max(x, y));
}

range operator|(const range &a, const range &b)
{
  return range(std::min(a.min(), b.min()), std::max(a.max(), b.max()));
}

range operator+(const range &a, const range &b)
{
  return range(a.min() + b.min(), a.max() + b.max());
}
```

```
range operator-(const range &a)
{
  return range(-a.max(), -a.min());
}

range operator-(const range &a, const range &b)
{
  return a + -b;
}

range operator/(const range &a, const int &b)
{
  // precondition: b is a power of two
  return range(a.min() / b, a.max() / b);
}

range operator*(const int &b, const range &a)
{
  // precondition: b is a power of two
  return range(b * a.min(), b * a.max());
}

range operator*(const range &a, const range &b)
{
  // pessimisation by 1ulp if multiplication is exact
  real x = a.min() * b.min();
  real y = a.min() * b.max();
  real z = a.max() * b.min();
  real w = a.max() * b.max();
  real mi = std::min(std::min(x, y), std::min(z, w));
  real ma = std::max(std::max(x, y), std::max(z, w));
  return range(mi, ma);
#if 0
    ( std::nextafter(mi, -infinity)
    , std::nextafter(ma,  infinity)
    );
#endif
}

range operator/(const range &x, const range &y)
{
  // pessimisation by 1ulp if division is exact
  if (0 <= y.min())
    return range(std::nextafter(x.min() / y.max(), -infinity), std::nextafter(x.↙
        ↳ max() / y.min(), infinity));
  else if (y.max() <= 0)
    return range(std::nextafter(x.min() / y.min(), -infinity), std::nextafter(x.↙
        ↳ max() / y.max(), infinity));
  else
    return range(-infinity, infinity); // division by 0
}

range sqr(const range &a)
{
  auto lo = a.min();
  auto hi = a.max();
```

```
  auto lo2 = sqr(lo);
  auto hi2 = sqr(hi);
  auto olo = std::min(lo2, hi2);
  auto ohi = std::max(lo2, hi2);
  if (lo <= 0 && 0 <= hi) olo = 0;
  return range(olo, ohi);
}

class box
{
  range b[2];
public:
  box() // empty
  {
    b[0] = range();
    b[1] = range();
  }
  box(const range &x, const range &y)
  {
    b[0] = x;
    b[1] = y;
  }
  const range &x() const { return b[0]; }
  const range &y() const { return b[1]; }
  bool operator<(const box &that) const
  {
    return
      x() != that.x()
    ? x() <  that.x()
    : y() != that.y()
    ? y() <  that.y()
    : false; // equal
  }
  bool operator!=(const box &that)
  {
    return *this < that || that < *this;
  }
};

bool inside(const range &a, const range &b)
{
  return b.min() < a.min() && a.max() < b.max();
}

bool inside(const box &a, const box &b)
{
  return
    inside(a.x(), b.x()) &&
    inside(a.y(), b.y());
}

box operator|(const box &a, const box &b)
{
  return box(a.x() | b.x(), a.y() | b.y());
}

typedef std::vector<box> (*function)(const box &, const box &);
```

```
typedef std::pair<function, function> bifunction;

std::vector<box> quadratic_forward(const box &zb, const box &cb)
{
  complex<range> z(zb.x(), zb.y());
  complex<range> c(cb.x(), cb.y());
  z = sqr(z) + c;
  std::vector<box> result;
  result.push_back(box(z.real(), z.imag()));
  return result;
}

range sqrt(const range &x)
{
  // 1ulp pessimistic if sqrt was exact
  return range
    ( std::nextafter(std::sqrt(x.min()), 0)
    , std::nextafter(std::sqrt(x.max()), infinity)
    );
}

range abs(const complex<range> &x)
{
  return sqrt(sqr(x.real()) + sqr(x.imag()));
}

complex<range> sqrt(const complex<range> &x)
{
  auto a = x.real();
  auto b = x.imag();
  if (b.min() == 0 && b.max() == 0)
  {
    if (a.min() >= 0)
      return complex<range>(sqrt(a), b);
    if (a.max() <= 0)
      return complex<range>(b, sqrt(-a));
  }
  if (a.min() == 0 && a.max() == 0)
  {
    if (b.min() >= 0)
    {
      auto c = sqrt( b / 2);
      return complex<range>(c,  c);
    }
    if (b.max() <= 0)
    {
      auto c = sqrt(-b / 2);
      return complex<range>(c, -c);
    }
  }
  auto r = abs(x);
  if (midpoint(a) >= 0)
  {
    auto t = r + a;
    auto u = sqrt(2 * t);
    return complex<range>(u / 2, b / u);
```

```
    }
    if (! contains(b, 0))
    {
      auto u = r - a;
      auto t = sqrt(2 * u);
      auto c = abs(b / t);
      auto d = t / 2;
      if (midpoint(b) < 0) d = -d;
      return complex<range>(c, d);
    }
    auto t = (r + a) / 2;
    auto u = (r - a) / 2;
    auto c = sqrt(t);
    auto d = sqrt(u);
    if (b.max() >= 0)
      ; // nop
    else if (b.min() >= 0)
      d = -d;
    else
      d = t | -t;
    return complex<range>(c, d);
  }

  std::vector<box> quadratic_reverse(const box &zb, const box &cb)
  {
    complex<range> z(zb.x(), zb.y());
    complex<range> c(cb.x(), cb.y());
    z = sqrt(z - c);
    std::vector<box> result;
    result.push_back(box( z.real(),  z.imag()));
    result.push_back(box(-z.real(), -z.imag()));
    return result;
  }


  const bifunction quadratic(quadratic_forward, quadratic_reverse);

  class cell_ref
  {
    std::vector<color> *lsb;
    size_t index;
    cell_ref(cell_ref &) = delete; // no copy
  public:
    cell_ref(std::vector<color> &lsb, size_t index)
    : lsb(&lsb), index(index)
    { }
    cell_ref(cell_ref &&c) noexcept // move
    : lsb(std::exchange(c.lsb, nullptr))
    , index(std::exchange(c.index, 0))
    {
    }
    cell_ref &operator=(const color &c)
    {
      (*lsb)[index] = c;
      return *this;
    }
    operator color () const
```

```
    {
      return (*lsb)[index];
    }
};

class const_cell_ref
{
  const std::vector<color> &lsb;
  size_t index;
  const_cell_ref(const_cell_ref &) = delete; // no copy
public:
  const_cell_ref(const std::vector<color> &lsb, size_t index)
  : lsb(lsb), index(index)
  { }
  const_cell_ref(const_cell_ref &&c) noexcept // move
  : lsb(std::move(c.lsb))
  , index(std::exchange(c.index, 0))
  {
  }
  operator color () const
  {
    return lsb[index];
  }
};

real lerp(const range &a, real x)
{
  // precondition: multiplications are exact
  return a.min() * (1 - x) + x * a.max();
}

range subrange(const range &a, real lo, real hi)
{
  return range(lerp(a, lo), lerp(a, hi));
}

std::pair<real, real> suprange(const range &bounds, const range &sub, size_t d)
{
  // precondition: all maths is exact
  real diameter = bounds.max() - bounds.min();
  real lo = std::floor((sub.min() - bounds.min()) / diameter * d);
  real hi = std::ceil ((sub.max() - bounds.min()) / diameter * d);
  return std::pair<real, real>(lo, hi);
}

class world
{
  box bounds;
  size_t n;
  std::vector<color> lsb;
  range cx, cy;
public:
  world(const box &bounds, size_t n, const range &cx, const range &cy) // 0
  : bounds(bounds)
  , n(n)
  , lsb((1LLU << (2 * n)) + 1)
  , cx(cx)
```

```
  , cy(cy)
455 {
    fill(gray);
    outside() = white;
  }
  world(const world &w, size_t k, size_t l)
460 : bounds(w.bounds)
  , n(w.n + 1)
  , lsb((1LLU << (2 * (w.n + 1))) + 1)
  , cx(subrange(w.cx, k / 2.0, (k + 1) / 2.0))
  , cy(subrange(w.cy, l / 2.0, (l + 1) / 2.0))
465 {
    for (size_t i = 0; i < 1LLU << n; ++i)
    for (size_t j = 0; j < 1LLU << n; ++j)
    {
      index a(i, j);
470   index b(i >> 1, j >> 1);
      cell(a) = inherit[w.cell(b)];
    }
    outside() = white;
  }
475
  world &fill(const color &c)
  {
    std::fill(lsb.begin(), lsb.end(), c);
    return *this;
480 }
  cell_ref cell(const index &ix)
  {
    return cell_ref(lsb, ix(n));
  }
485 color cell(const index &ix) const
  {
    return const_cell_ref(lsb, ix(n));
  }
  cell_ref outside()
490 {
    return cell_ref(lsb, 1LLU << (2 * n));
  }

  color outside() const
495 {
    return const_cell_ref(lsb, 1LLU << (2 * n));
  }

  box box_for(const index &a)
500 {
    real d = 1LLU << n;
    return box
      ( subrange(bounds.x(), a.x / d, (a.x + 1) / d)
      , subrange(bounds.y(), a.y / d, (a.y + 1) / d)
505   );
  }

  std::vector<index> cells_for(const box &b, real expand = 0)
  {
510   size_t d = 1LLU << n;
```

```
    std::vector<index> result;
    if (! (inside(b, bounds))) result.push_back(index(1LLU << n, 0)); // ↙
        ↳ exterior
    std::pair<real, real> x = suprange(bounds.x(), b.x(), d);
    std::pair<real, real> y = suprange(bounds.y(), b.y(), d);
    for (real i = x.first - expand; i <= x.second + expand; i += 1)
    {
      if (i < 0) continue;
      if (i >= d) break;
      for (real j = y.first - expand; j <= y.second + expand; j += 1)
      {
        if (j < 0) continue;
        if (j >= d) break;
        result.push_back(index(size_t(i), size_t(j)));
      }
    }
    return result;
  }

  bool whiten(const bifunction &f)
  {
    bool any_changed = false;
    for (size_t i = 0; i < 1LLU << n; ++i)
    for (size_t j = 0; j < 1LLU << n; ++j)
    {
      index a(i, j);
      any_changed |= whiten(f, a);
    }
    return any_changed;
  }

  bool whiten(const bifunction &f, const index &a)
  {
    const box c0(cx, cy);
    bool any_changed = false;
    if (cell(a) == gray)
    {
      auto me = box_for(a);
      bool all_white = true;
      for (const auto &b : f.first(me, c0))
      {
        for (const auto &c : cells_for(b))
        {
          all_white &= (cell(c) == white);
          if (! all_white) break;
        }
        if (! all_white) break;
      }
      if (all_white)
      {
        cell(a) = white;
        any_changed |= true;
#if 0
        for (const auto &b : f.second(me, c0))
        {
          for (const auto &c : cells_for(b))
          {
```

```
            any_changed |= whiten(f, c);
          }
        }
#endif
      }
    }
    return any_changed;
  }

  bool blacken(const bifunction &f)
  {
    bool any_changed = false;
    for (size_t i = 0; i < 1LLU << n; ++i)
    for (size_t j = 0; j < 1LLU << n; ++j)
    {
      index a(i, j);
      any_changed |= blacken(f, a);
    }
    return any_changed;
  }

  bool blacken(const bifunction &f, const index &a)
  {
    const box c0(cx, cy);
    bool any_changed = false;
    if (cell(a) == gray)
    {
      auto me = box_for(a);
      bool any_white_or_marked = false;
      for (const auto &b : f.first(me, c0))
      {
        for (const auto &c : cells_for(b, 1))
        {
          any_white_or_marked |= (cell(c) == white || cell(c) == marked);
          if (any_white_or_marked) break;
        }
        if (any_white_or_marked) break;
      }
      if (any_white_or_marked)
      {
        cell(a) = marked;
        any_changed |= true;
#if 0
        for (const auto &b : f.second(me, c0))
        {
          for (const auto &c : cells_for(b, 1))
          {
            any_changed |= blacken(f, c);
          }
        }
#endif
      }
    }
    return any_changed;
  }

  void unmark()
```

```
  {
    for (size_t i = 0; i < 1LLU << n; ++i)
    for (size_t j = 0; j < 1LLU << n; ++j)
    {
      index a(i, j);
      cell(a) = unmarked[cell(a)];
    }
  }

  bool step(const bifunction &f)
  {
    const std::vector<color> lsb_backup = lsb;
    while (whiten(f))
      ;
    while (blacken(f))
      ;
    unmark();
    // gray -> marked -> gray is possible, so do it brute
    return lsb_backup != lsb;
  }

  std::string ppm_header()
  {
    std::ostringstream os;
    os << "P6\n" << (1LLU << n) << " " << (1LLU << n) << "\n# "
       << cx.min() << "  " << cx.max() << "\n# "
       << cy.min() << "  " << cy.max() << "\n255\n";
    return os.str();
  }

  std::vector<unsigned char> raster()
  {
    std::vector<unsigned char> rgb(3LLU << (2 * n));
    for (size_t i = 0; i < 1LLU << n; ++i)
    for (size_t j = 0; j < 1LLU << n; ++j)
    {
      size_t ix = 3 * ((i << n) + j);
      const color c = cell(index(i, j));
      rgb[ix++] = colorrgb[c][0];
      rgb[ix++] = colorrgb[c][1];
      rgb[ix++] = colorrgb[c][2];
    }
    return rgb;
  }

};

size_t z_order(size_t x, size_t y, size_t z, size_t w)
{
#define CHAR_BIT 8
  size_t o = 0;
  for (size_t i = 0; i < sizeof(x) * CHAR_BIT / 4; i++)
  {
    o |= (x & 1LLU << i) << (3 * i + 0)
      |  (y & 1LLU << i) << (3 * i + 1)
      |  (z & 1LLU << i) << (3 * i + 2)
      |  (w & 1LLU << i) << (3 * i + 3)
```

```
      ;
  }
  return o;
#undef CHAR_BIT
}

void depth_first(size_t &counter, unsigned char *voxels[2], world &w, size_t ↙
    ↳ max_n, size_t n, size_t x, size_t y)
{
  while (w.step(quadratic))
    ;
  if (n == max_n)
  {
    counter += 1;
    std::cerr << "\t" << counter << "\r";
    for (size_t i = 0; i < 1LLU << n; ++i)
    for (size_t j = 0; j < 1LLU << n; ++j)
    {
      const color c = w.cell(index(i, j));
      const size_t ix = z_order(x, y, i, j);
      const size_t byte = ix >> 3;
      const size_t bit = ix & 7;
      voxels[0][byte] |= (c == black ? 1 : 0) << bit;
      voxels[1][byte] |= (c == gray  ? 1 : 0) << bit;
    }
  }
  if (n <  max_n)
  {
    for (size_t k = 0; k < 2; ++k)
    for (size_t l = 0; l < 2; ++l)
    {
      auto w2 = world(w, k, l);
      depth_first(counter, voxels, w2, max_n, n + 1, (x << 1) + k, (y << 1) + l)↙
          ↳ ;
    }
  }
}


int main(int argc, char **argv)
{
  (void) argc;
  (void) argv;
  range r(-2, 2);
  box bounds(r, r);
  world w(bounds, 0, r, r);
  size_t n = 9;
  size_t bytes = 1LLU << (4 * n - 3);
  unsigned char *voxels[2] =
    { (unsigned char *) calloc(1, bytes)
    , (unsigned char *) calloc(1, bytes)
    };
  size_t counter = 0;
  depth_first(counter, voxels, w, n, 0, 0, 0);
  std::FILE *f;
  f = std::fopen("black.raw", "wb");
  std::fwrite(voxels[0], bytes, 1, f);
```

```
  std::fclose(f);
  f = std::fopen("gray.raw", "wb");
  std::fwrite(voxels[1], bytes, 1, f);
  std::fclose(f);
  return 0;
}
```

57 ultimate-anti-buddha/anti.c

```
// Ultimate Anti-Buddhabrot (c) 2013 Claude Heiland-Allen
// https://mathr.co.uk/blog/2013-12-30_ultimate_anti-buddhabrot.html
// gcc -std=c99 -Wall -pedantic -Wextra -O3 -fopenmp -lm -o anti anti.c
// ./anti > anti.ppm

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


typedef unsigned int N;
typedef unsigned long int NN;
typedef int Z;
typedef long int ZZ;
typedef double R;
typedef double complex C;
typedef float F;


static const R pi = 3.141592653589793;


static const Z zwidth = 4096;
static const Z zheight = 4096;
static const C zpixel_size = 5.0 / 4096;
static const C z0 = 0;


static F count[4096][4096][4];
static ZZ total = 0;


R cabs2(C z) { return creal(z) * creal(z) + cimag(z) * cimag(z); }


static void hsv2rgb(F h, F s, F v, F *rp, F *gp, F *bp) {
  F i, f, p, q, t, r, g, b;
  Z ii;
  if (s == 0.0) {
    r = v;
    g = v;
    b = v;
  } else {
    h = h - floor(h);
    h = h * 6.0;
    i = floor(h);
```

```
      ii = i;
      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;
      }
    }
    *rp = r;
    *gp = g;
    *bp = b;
  }


  static void plot_iterates(C z1, C c, N period, R r, R g, R b) {
    for (N j = 0; j < 2; ++j) {
      C z = z1;
      for (N i = 0; i < period; ++i) {
        C pz0 = (zwidth / 2.0 - 0.5) + I * (zheight / 2.0 - 0.5);
        C pz = (z - z0) / zpixel_size + pz0;
        Z x = creal(pz);
        Z y = cimag(pz);
        if (0 <= x && x < zwidth && 0 <= y && y < zheight) {
          #pragma omp atomic update
          count[y][x][0] += r;
          #pragma omp atomic update
          count[y][x][1] += g;
          #pragma omp atomic update
          count[y][x][2] += b;
          #pragma omp atomic update
          count[y][x][3] += 1;
          #pragma omp atomic update
          total++;
        }
        z = z * z + c;
      }
      z1 = conj(z1);
      c = conj(c);
    }
  }


  static void clear_image() {
    memset(&count[0][0][0], 0, zwidth * zheight * 4 * sizeof(F));
    total = 0;
  }


  static void output_image() {
    R s = zwidth * zheight / (R) total;
    printf("P6\n%u %u\n255\n", zwidth, zheight);
```

```
  for (Z y = 0; y < zheight; ++y) {
    for (Z x = 0; x < zwidth; ++x) {
      if (count[y][x][3]) {
        R v = log2(1 + count[y][x][3] * s);
        for (Z c = 0; c < 3; ++c) {
          R u = count[y][x][c] / count[y][x][3];
          Z o = fminf(fmaxf(54 * v * u, 0), 255);
          putchar(o);
        }
      } else {
        putchar(0);
        putchar(0);
        putchar(0);
      }
    }
  }
}


int wucleus(C *z0, C c, N period) {
  R eps = 1e-12;
  R er2 = 16;
  C z = *z0;
  for (N j = 0; j < 256; ++j) {
    C dz = 1.0;
    for (N k = 0; k < period; ++k) {
      dz = 2.0 * dz * z;
      z = z * z + c;
    }
    R z2 = cabs(z);
    if (! (z2 < er2)) {
      break;
    }
    z = *z0 - (z - *z0) / (dz - 1.0);
    R e = cabs(z - *z0);
    *z0 = z;
    if (e < eps) {
      return 1;
    }
  }
  return 0;
}

R interior_distance(C *w, C c, N period, R pixel_size) {
  if (wucleus(w, c, period)) {
    C z = *w;
    C dz = 1.0;
    C dzdz = 0.0;
    C dc = 0.0;
    C dcdz = 0.0;
    for (N j = 0; j < period; ++j) {
      dcdz = 2.0 * (z * dcdz + dz * dc);
      dc = 2.0 * z * dc + 1.0;
      dzdz = 2.0 * (dz * dz + z * dzdz);
      dz = 2.0 * z * dz;
      z = z * z + c;
```

```
    }
    return (1.0 - cabs2(dz)) / (cabs(dcdz + dzdz * dc / (1.0 - dz)) * pixel_size↙
        ↳ );
  }
  return -1.0;
}

static void render_recursive_interior(F red, F grn, F blu, N period, C z, C c, R↙
    ↳ grid_spacing, N depth) {
  if (depth == 0) { return; }
  C z1 = z;
  R de = interior_distance(&z1, c, period, grid_spacing);
  if (de > 0) {
    plot_iterates(z1, c, period, red, grn, blu);
    render_recursive_interior(red, grn, blu, period, z1, c + 1 * 0.5 * ↙
        ↳ grid_spacing, 0.5 * grid_spacing, depth - 1);
    render_recursive_interior(red, grn, blu, period, z1, c - 1 * 0.5 * ↙
        ↳ grid_spacing, 0.5 * grid_spacing, depth - 1);
    render_recursive_interior(red, grn, blu, period, z1, c + I * 0.5 * ↙
        ↳ grid_spacing, 0.5 * grid_spacing, depth - 1);
    render_recursive_interior(red, grn, blu, period, z1, c - I * 0.5 * ↙
        ↳ grid_spacing, 0.5 * grid_spacing, depth - 1);
  }
}

static void render_recursive_unknown(C c, R grid_spacing, N depth) {
  if (depth == 0) { return; }
  C z = 0;
  C dz = 0;
  R mz2 = 1.0/0.0;
  N maxiters = 1 << 12;
  for (N i = 1; i < maxiters; ++i) {
    dz = 2 * z * dz + 1;
    z = z * z + c;
    R z2 = cabs2(z);
    if (! (z2 < 65536)) {
      R de = sqrt(z2) * log(z2) / (cabs(dz) * grid_spacing);
      if (de < 4) {
        render_recursive_unknown(c + 1 * 0.5 * grid_spacing, 0.5 * grid_spacing,↙
            ↳ depth - 1);
        render_recursive_unknown(c - 1 * 0.5 * grid_spacing, 0.5 * grid_spacing,↙
            ↳ depth - 1);
        render_recursive_unknown(c + I * 0.5 * grid_spacing, 0.5 * grid_spacing,↙
            ↳ depth - 1);
        render_recursive_unknown(c - I * 0.5 * grid_spacing, 0.5 * grid_spacing,↙
            ↳ depth - 1);
      }
      break;
    }
    if (z2 < mz2) {
      mz2 = z2;
      C z1 = z;
      R de = interior_distance(&z1, c, i, grid_spacing);
      if (de > 0) {
        R phi5 = pow((sqrt(5) + 1) / 2, 5);
        R hue = i / phi5 ;
        F red, grn, blu;
```

```
210           hsv2rgb(hue, 1, 1, &red, &grn, &blu);
              plot_iterates(z1, c, i, red, grn, blu);
              if (de > 0.25) {
                render_recursive_interior(red, grn, blu, i, z1, c + 1 * 0.5 * ↙
                    ↳ grid_spacing, 0.5 * grid_spacing, depth - 1);
                render_recursive_interior(red, grn, blu, i, z1, c - 1 * 0.5 * ↙
                    ↳ grid_spacing, 0.5 * grid_spacing, depth - 1);
215             render_recursive_interior(red, grn, blu, i, z1, c + I * 0.5 * ↙
                    ↳ grid_spacing, 0.5 * grid_spacing, depth - 1);
                render_recursive_interior(red, grn, blu, i, z1, c - I * 0.5 * ↙
                    ↳ grid_spacing, 0.5 * grid_spacing, depth - 1);
              } else {
                render_recursive_unknown(c + 1 * 0.5 * grid_spacing, 0.5 * ↙
                    ↳ grid_spacing, depth - 1);
                render_recursive_unknown(c - 1 * 0.5 * grid_spacing, 0.5 * ↙
                    ↳ grid_spacing, depth - 1);
220             render_recursive_unknown(c + I * 0.5 * grid_spacing, 0.5 * ↙
                    ↳ grid_spacing, depth - 1);
                render_recursive_unknown(c - I * 0.5 * grid_spacing, 0.5 * ↙
                    ↳ grid_spacing, depth - 1);
              }
              break;
            }
225       }
    }
  }

  static void render() {
230   clear_image();
    N grid = 1024;
    R grid_spacing = 5.0 / grid;
    #pragma omp parallel for schedule(dynamic, 1)
    for (N y = 0; y < grid; ++y) {
235   for (N x = 0; x < grid; ++x) {
        C c = grid_spacing * ((x + 0.5 - grid/2.0) + I * (y + 0.5 - grid/2.0));
        if (cimag(c) >= 0) {
          render_recursive_unknown(c, grid_spacing, 9);
        }
240     }
      #pragma omp critical
      fprintf(stderr, "%8d\r", y);
    }
    output_image();
245 }


  extern int main(int argc, char **argv) {
    (void) argc;
250 (void) argv;
    render();
    return 0;
  }
```

58 ultimate-anti-buddha/geometry.h

```
#ifndef GEOMETRY_H
#define GEOMETRY_H 1
```

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

/*═══════════════════════════════════════════════════════════════════════*\
** Vector and Matrix functions                                           **
\*═══════════════════════════════════════════════════════════════════════*/

/* 4-vector */
struct vector4 {
  double v[4];
};

/* 4x4-matrix */
struct matrix4 {
  double m[16];
};

/* 3-vector */
struct vector3 {
  double v[3];
};

/* matrix := diag(*f) */
static inline void mdiag4(struct matrix4 *m, const double *f) {
  memset(m, 0, sizeof(struct matrix4));
  double *n = m->m;
  n[ 0] = f[0];
  n[ 5] = f[1];
  n[10] = f[2];
  n[15] = f[3];
}

/* matrix := rotation(i,j,angle) */
static inline void mrot4(struct matrix4 *m, int i, int j, double a) {
  memset(m, 0, sizeof(struct matrix4));
  double *n = m->m;
  n[ 0] = 1.0;
  n[ 5] = 1.0;
  n[10] = 1.0;
  n[15] = 1.0;
  double c = cos(a);
  double s = sin(a);
  n[i*5]   =  c;
  n[j*5]   =  c;
  n[i*4+j] =  s;
  n[j*4+i] = -s;
}

/* matrix := matrix * matrix */
static inline void mmul4m(
  struct matrix4 *m,
  const struct matrix4 *l,
  const struct matrix4 *r
) {
  for (int i = 0; i < 4; ++i) {
    for (int j = 0; j < 4; ++j) {
```

```
        double f = 0.0;
        for (int k = 0; k < 4; ++k) {
          f += l->m[i*4+k] * r->m[k*4+j];
        }
        m->m[i*4+j] = f;
      }
    }
  }

  /* vector := matrix * vector */
  static inline void mmul4v(
    struct vector4 *v,
    const struct matrix4 *l,
    const struct vector4 *r
  ) {
    for (int i = 0; i < 4; ++i) {
      double f = 0.0;
      for (int j = 0; j < 4; ++j) {
        f += l->m[i*4+j] * r->v[j];
      }
      v->v[i] = f;
    }
  }

  /*=========================================================================*\
  ** Quaternion functions                                                    **
  ** ----------------------------------------------------------------------**
  ** http://www.lce.hut.fi/~ssarkka/pub/quat.pdf                            **
  \*=========================================================================*/

  /* quaternion data structure */
  struct quaternion {
    double q[4];
  };

  /* quaternion norm squared */
  static inline double qnorm2(const struct quaternion *p) {
    double p1, p2, p3, p4, r;
    p1 = p->q[0]; p2 = p->q[1]; p3 = p->q[2]; p4 = p->q[3];
    r = p1*p1 + p2*p2 + p3*p3 + p4*p4;
    return r;
  }

  static inline double qdot(const struct quaternion *p, const struct quaternion *q
      ) {
    double p1, p2, p3, p4, q1, q2, q3, q4, r;
    p1 = p->q[0]; p2 = p->q[1]; p3 = p->q[2]; p4 = p->q[3];
    q1 = q->q[0]; q2 = q->q[1]; q3 = q->q[2]; q4 = q->q[3];
    r = p1*q1 + p2*q2 + p3*q3 + p4*q4;
    return r;
  }

  /* quaternion norm */
  static inline double qnorm(const struct quaternion *p) {
    return sqrt(qnorm2(p));
  }
```

```
/* quaternion conjugate: r := p* */
static inline void qconj(struct quaternion *r, const struct quaternion *p) {
  r->q[0] =  p->q[0];
  r->q[1] = -p->q[1];
  r->q[2] = -p->q[2];
  r->q[3] = -p->q[3];
}

/* quaternion inverse: r := p^-1 */
/* p^-1 = p* / |p|^2 */
static inline void qinv(struct quaternion *r, const struct quaternion *p) {
  double pn2 = qnorm2(p);
  r->q[0] =  p->q[0] / pn2;
  r->q[1] = -p->q[1] / pn2;
  r->q[2] = -p->q[2] / pn2;
  r->q[3] = -p->q[3] / pn2;
}

static inline void qneg(struct quaternion *r, const struct quaternion *p) {
  r->q[0] = -p->q[0];
  r->q[1] = -p->q[1];
  r->q[2] = -p->q[2];
  r->q[3] = -p->q[3];
}

/* quaternion addition: r := q + p */
static inline void qadd(
  struct quaternion *r,
  const struct quaternion *q,
  const struct quaternion *p
) {
  double q1, q2, q3, q4, p1, p2, p3, p4, r1, r2, r3, r4;
  q1 = q->q[0]; q2 = q->q[1]; q3 = q->q[2]; q4 = q->q[3];
  p1 = p->q[0]; p2 = p->q[1]; p3 = p->q[2]; p4 = p->q[3];
  r1 = q1 + p1;
  r2 = q2 + p2;
  r3 = q3 + p3;
  r4 = q4 + p4;
  r->q[0] = r1; r->q[1] = r2; r->q[2] = r3; r->q[3] = r4;
}

/* quaternion multiplication: r := q * p */
static inline void qmul(
  struct quaternion *r,
  const struct quaternion *q,
  const struct quaternion *p
) {
  double q1, q2, q3, q4, p1, p2, p3, p4, r1, r2, r3, r4;
  q1 = q->q[0]; q2 = q->q[1]; q3 = q->q[2]; q4 = q->q[3];
  p1 = p->q[0]; p2 = p->q[1]; p3 = p->q[2]; p4 = p->q[3];
  r1 = q1 * p1 - q2 * p2 - q3 * p3 - q4 * p4;
  r2 = q2 * p1 + q1 * p2 - q4 * p3 + q3 * p4;
  r3 = q3 * p1 + q4 * p2 + q1 * p3 - q2 * p4;
  r4 = q4 * p1 - q3 * p2 + q2 * p3 + q1 * p4;
  r->q[0] = r1; r->q[1] = r2; r->q[2] = r3; r->q[3] = r4;
}
```

```
/* quaternion exponentiation: r := exp(p) */
/* exp(s;v) = exp(s) (cos(|v|) ; v sin(|v|) / |v|) */
static inline void qexp(struct quaternion *r, const struct quaternion *p) {
  double s, v0, v1, v2, vn, se, vns, vnc;
  s   = p->q[0];
  v0  = p->q[1];
  v1  = p->q[2];
  v2  = p->q[3];
  vn  = sqrt(v0*v0 + v1*v1 + v2*v2);
  vnc = cos(vn);
  vns = sin(vn) / vn;
  se  = exp(s);
  r->q[0] = se * vnc;
  r->q[1] = v0 * vns;
  r->q[2] = v1 * vns;
  r->q[3] = v2 * vns;
}

/* quaternion logarithm: r := log(p) */
/* ln(q@(s;v)) = (ln(|q|) ; v acos(s / |q|) / |v|) */
static inline void qlog(struct quaternion *r, const struct quaternion *p) {
  double s, v0, v1, v2, vn2, vn, qn, qnl, sac;
  s   = p->q[0];
  v0  = p->q[1];
  v1  = p->q[2];
  v2  = p->q[3];
  vn2 = v0*v0 + v1*v1 + v2*v2;
  vn  = sqrt(vn2);
  qn  = sqrt(s*s + vn2);
  qnl = log(qn);
  sac = acos(s / qn) / vn;
  r->q[0] = qnl;
  r->q[1] = v0 * sac;
  r->q[2] = v1 * sac;
  r->q[3] = v2 * sac;
}

/* quaternion scalar power: r := p^f */
/* p^f = exp(ln(p) f) */
static inline void qpows(
  struct quaternion *r,
  const struct quaternion *p,
  double f
) {
  struct quaternion l;
  qlog(&l, p);
  l.q[0] *= f;
  l.q[1] *= f;
  l.q[2] *= f;
  l.q[3] *= f;
  qexp(r, &l);
}

/* quaternion slerp: r := Slerp(p,q,t) */
/* Slerp(p,q;t) = (q p^-1)^t p */
static inline void qslerp(
  struct quaternion *r,
```

```
    const struct quaternion *p,
    const struct quaternion *q,
    double t
  ) {
    struct quaternion qq, p1, qp1, qp1t;
#if 0
    if (qdot(p, q) < 0)
    {
      qneg(&qq, q);
    }
    else
#endif
    {
      qq = *q;
    }
    qinv(&p1, p);
    qmul(&qp1, &qq, &p1);
    qpows(&qp1t, &qp1, t);
    qmul(r, &qp1t, p);
  }

  /*=============================================================================*\
  ** Quaternions to 4D Rotation Matrix                                           **
  ** ------------------------------------------------------------------------- **
  ** http://www.cs.indiana.edu/pub/techreports/TR406.pdf                         **
  \*=============================================================================*/

  /* 4x4 rotation matrix from two unit(?) quaternions */
  static inline void mquat2(
    struct matrix4 *m,
    const struct quaternion *q,
    const struct quaternion *p
  ) {
    double q0, q1, q2, q3, p0, p1, p2, p3;
    q0 = q->q[0]; q1 = q->q[1]; q2 = q->q[2]; q3 = q->q[3];
    p0 = p->q[0]; p1 = p->q[1]; p2 = p->q[2]; p3 = p->q[3];

    m->m[ 0] =   q0*p0 + q1*p1 + q2*p2 + q3*p3;
    m->m[ 1] =   q1*p0 - q0*p1 - q3*p2 + q2*p3;
    m->m[ 2] =   q2*p0 + q3*p1 - q0*p2 - q1*p3;
    m->m[ 3] =   q3*p0 - q2*p1 + q1*p2 - q0*p3;

    m->m[ 4] = - q1*p0 + q0*p1 - q3*p2 + q2*p3;
    m->m[ 5] =   q0*p0 + q1*p1 - q2*p2 - q3*p3;
    m->m[ 6] = - q3*p0 + q2*p1 + q1*p2 - q0*p3;
    m->m[ 7] =   q2*p0 + q3*p1 + q0*p2 + q1*p3;

    m->m[ 8] = - q2*p0 + q3*p1 + q0*p2 - q1*p3;
    m->m[ 9] =   q3*p0 + q2*p1 + q1*p2 + q0*p3;
    m->m[10] =   q0*p0 - q1*p1 + q2*p2 - q3*p3;
    m->m[11] = - q1*p0 - q0*p1 + q3*p2 + q2*p3;

    m->m[12] = - q3*p0 - q2*p1 + q1*p2 + q0*p3;
    m->m[13] = - q2*p0 + q3*p1 - q0*p2 + q1*p3;
    m->m[14] =   q1*p0 + q0*p1 + q3*p2 + q2*p3;
    m->m[15] =   q0*p0 - q1*p1 - q2*p2 + q3*p3;
```

```
}


static inline void mqslerp2(
  struct matrix4 *m,
  const struct quaternion *q0,
  const struct quaternion *p0,
  const struct quaternion *q1,
  const struct quaternion *p1,
  double t
) {
  struct quaternion p, q;
  qslerp(&q, q0, q1, t);
  qslerp(&p, p0, p1, t);
  mquat2(m, &q, &p);
}

/* generate vectors in cube, discard those outside sphere, normalize */
#define randf() (2.0 * (double) rand() / (double) RAND_MAX - 1.0)
static inline void random_unit(double *v, const int n) {
  double r = 0.0;
  while (r < 0.00001 || 1.0 < r) {
    r = 0.0;
    for (int i = 0; i < n; ++i) {
      v[i] = randf();
      r += v[i] * v[i];
    }
  }
  r = sqrt(r);
  for (int i = 0; i < n; ++i) {
    v[i] /= r;
  }
}
#undef randf

static inline void random_unit_quaternion(struct quaternion *q)
{
  random_unit(&q->q[0], 4);
}

#endif
```

59 ultimate-anti-buddha/.gitignore

```
anti
UltimateAntiBuddhagram
```

60 ultimate-anti-buddha/Makefile

```
UltimateAntiBuddhagram: UltimateAntiBuddhagram.c geometry.h
        gcc -std=c99 -Wall -pedantic -Wextra -O3 -fopenmp -lm -o ↙
            ↳ UltimateAntiBuddhagram UltimateAntiBuddhagram.c -g -lGL -lGLEW -↙
            ↳ lglfw

anti: anti.c
        gcc -std=c99 -Wall -pedantic -Wextra -O3 -fopenmp -lm -o anti anti.c
```

61 ultimate-anti-buddha/UltimateAntiBuddhagram.c

```
// Ultimate Anti-Buddhagram (c) 2019 Claude Heiland-Allen
// ./UltimateAntiBuddhagram | ffmpeg -i - video.mp4

#define _GNU_SOURCE
#define _FILE_OFFSET_BITS 64
#include <sys/mman.h>
#include <fcntl.h>
#include <unistd.h>
#include <errno.h>
#ifndef MAP_HUGE_2MB
#define MAP_HUGE_2MB (21 << MAP_HUGE_SHIFT)
#endif
#ifndef MAP_HUGE_1GB
#define MAP_HUGE_1GB (30 << MAP_HUGE_SHIFT)
#endif

#include <complex.h>
#include <math.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <GL/glew.h>
#include <GLFW/glfw3.h>

#include "geometry.h"

static inline int sgn(double x) { return (x > 0) - (0 > x); }

static void *malloc_huge(size_t *size)
{
  // round up to 1GB
  size_t v = *size;
  v += (1 << 30) - 1;
  v >>= 30;
  v <<= 30;
  *size = v;
  size_t bytes = v;
  void *memory = mmap(NULL, bytes, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_HUGETLB | MAP_HUGE_1GB | MAP_ANONYMOUS, -1, 0);
  if (memory == MAP_FAILED)
  {
    fprintf(stderr, "could not mmap %lu with 1GB pages: %d\n", bytes, errno);
    memory = mmap(NULL, bytes, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_HUGETLB | MAP_HUGE_2MB | MAP_ANONYMOUS, -1, 0);
    if (memory == MAP_FAILED)
    {
      fprintf(stderr, "could not mmap %lu with 2MB pages: %d\n", bytes, errno);
      memory = mmap(NULL, bytes, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, -1, 0);
      if (memory == MAP_FAILED)
      {
        fprintf(stderr, "could not mmap %lu at all: %d\n", bytes, errno);
        return NULL;
```

```
      }
      else
      {
        fprintf(stderr, "did mmap %lu with 4kB pages\n", bytes);
      }
    }
    else
    {
      fprintf(stderr, "did mmap %lu with 2MB pages\n", bytes);
    }
  }
  else
  {
    fprintf(stderr, "did mmap %lu with 1GB pages\n", bytes);
  }
  return memory;
}

static void free_huge(void *memory, size_t size)
{
  munmap(memory, size);
}

typedef unsigned int N;
typedef unsigned long long int NN;
typedef int Z;
typedef long long int ZZ;
typedef double R;
typedef double complex C;
typedef float F;
typedef unsigned char B;

typedef struct { R x[5]; } V5; // z, c, 1
static const V5 v0 = { { -0.260365638451579340, 0, -0.368941235857094540, 0, 0 }↙
    ↪ };
typedef struct { R x[5][5]; } M55;

static inline M55 mul_m55_m55(const M55 *a, const M55 *b) {
  M55 o;
  for (Z i = 0; i < 5; ++i)
    for (Z j = 0; j < 5; ++j)
    {
      o.x[i][j] = 0;
      for (Z k = 0; k < 5; ++k)
        o.x[i][j] += a->x[i][k] * b->x[k][j];
    }
  return o;
}

static inline V5 mul_m55_v5(const M55 *a, const V5 *b) {
  V5 o = { { 0, 0, 0, 0, 0 } };
  for (Z i = 0; i < 5; ++i)
    for (Z j = 0; j < 5; ++j)
      o.x[i] += a->x[i][j] * b->x[j];
  return o;
}
```

```
typedef unsigned short int P;

#define DIM 13
typedef struct
{
  // c is implicit from coordinates
  // only one half of the set is stored due to symmetry
  C z[1 << (DIM - 1)][1 << DIM];
  P p[1 << (DIM - 1)][1 << DIM];
} cloud; // 9 GB
static inline C cloud_c(Z i, Z j, Z n)
{
  double x = ((i + 0.5) / (1 << n) - 0.5) * 4;
  double y = ((j + 0.5) / (1 << n) - 0.5) * 4;
  return x + I * y;
}

cloud *cloud_alloc(size_t *bytes)
{
  *bytes = sizeof(cloud);
  return malloc_huge(bytes);
}

void cloud_free(cloud *K, size_t bytes)
{
  free_huge(K, bytes);
}

void cloud_clear(cloud *K)
{
  memset(K, 0, sizeof(*K));
}

typedef struct { F x[4]; } V4;
#define MAXPERIOD (1 << 12)
typedef struct
{
  V4 *cz[MAXPERIOD];
  ZZ n[MAXPERIOD];
  ZZ i[MAXPERIOD];
  size_t bytes;
  size_t bytes_used;
} points;
static points *points_alloc(ZZ total)
{
  points *Q = calloc(1, sizeof(*Q));
  if (! Q)
    return 0;
  Q->bytes = total * sizeof(V4);
  Q->bytes_used = Q->bytes;
  Q->cz[0] = malloc_huge(&Q->bytes);
  if (! Q->cz[0])
  {
    free(Q);
    return 0;
  }
  return Q;
```

```
}
static void points_free_data(points *Q)
{
  free_huge(Q->cz[0], Q->bytes);
  memset(Q->cz, 0, sizeof(Q->cz));
  Q->bytes = 0;
}

#define WIDTH (1920)
#define HEIGHT (1080)

#if 0
typedef struct { F a[HEIGHT][WIDTH][4]; } accum;
accum *accum_alloc(void) { return malloc(sizeof(accum)); }
void accum_free(accum *A) { free(A); }
void accum_clear(accum *A) { memset(A, 0, sizeof(*A)); }
static inline void accum_plot(accum *A, R x0, R y0, F r, F g, F b, F a)
{
  R s[2] = { 1 - (x0 - floor(x0)), x0 - floor(x0) };
  R t[2] = { 1 - (y0 - floor(y0)), y0 - floor(y0) };
  if (-1 < x0 && x0 < WIDTH && -1 < y0 && y0 < HEIGHT)
  {
    for (Z j = 0; j < 2; ++j)
    {
      Z y = floor(y0) + j;
      if (0 <= y && y < HEIGHT)
      {
        for(Z i = 0; i < 2; ++i)
        {
          Z x = floor(x0) + i;
          if (0 <= x && x < WIDTH)
          {
            #pragma omp atomic update
            A->a[y][x][0] += s[i] * t[j] * a * r;
            #pragma omp atomic update
            A->a[y][x][1] += s[i] * t[j] * a * g;
            #pragma omp atomic update
            A->a[y][x][2] += s[i] * t[j] * a * b;
            #pragma omp atomic update
            A->a[y][x][3] += s[i] * t[j];
          }
        }
      }
    }
  }
}
#endif

typedef struct { B g[HEIGHT][WIDTH][3]; } image;
image *image_alloc(void) { return malloc(sizeof(image)); }
void image_free(image *G) { free(G); }

#if 0
void image_from_accum(image *G, const accum *A, ZZ total)
{
  R s = WIDTH * HEIGHT / (R) total;
  #pragma omp parallel for
```

```
  for (Z y = 0; y < HEIGHT; ++y) {
    for (Z x = 0; x < WIDTH; ++x) {
      F a = A->a[y][x][3];
      if (a) {
        R v = log2(1 + a * s) / a;
        for (Z c = 0; c < 3; ++c) {
          R u = A->a[y][x][c];
          Z o = fminf(fmaxf(255 * v * u, 0), 255);
          G->g[y][x][c] = 255 - o;
        }
      } else {
        for (Z c = 0; c < 3; ++c) {
          G->g[y][x][c] = 255;
        }
      }
    }
  }
}
#endif

void image_write(FILE *f, const image *G)
{
  fprintf(f, "P6\n%d %d\n255\n", WIDTH, HEIGHT);
  fwrite(&G->g[0][0][0], WIDTH * HEIGHT * 3, 1, f);
  fflush(f);
}

static inline R cnorm(C z) { return creal(z) * creal(z) + cimag(z) * cimag(z); }

static void hsv2rgb(F h, F s, F v, F *rp, F *gp, F *bp) {
  F i, f, p, q, t, r, g, b;
  Z ii;
  if (s == 0.0) {
    r = v;
    g = v;
    b = v;
  } else {
    h = h - floor(h);
    h = h * 6.0;
    i = floor(h);
    ii = i;
    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;
    }
  }
  *rp = r;
  *gp = g;
  *bp = b;
```

```
}

#if 0
static ZZ accum_from_cloud(accum *A, const cloud *K, const M55 *T1, const M55 * ↙
    ↳ T2)
{
  ZZ t = 0;
  #pragma omp parallel for schedule(dynamic, 1) reduction(+:t)
  for (Z j = 0; j < 1 << (DIM - 1); ++j)
  {
    for (Z i = 0; i < 1 << DIM; ++i)
    {
      P p = K->p[j][i];
      R phi5 = pow((sqrt(5) + 1) / 2, 5);
      R hue = (p - 1) / phi5;
      F r, g, b;
      hsv2rgb(hue, 0.5, 1, &r, &g, &b);
      for (Z k = 0; k < 2; ++k)
      {
        C c = cloud_c(i, j, DIM);
        C z = K->z[j][i];
        if (k)
        {
          c = conj(c);
          z = conj(z);
        }
        for (P n = 0; n < p; ++n)
        {
          V5 u = { { creal(c), cimag(c), creal(z), cimag(z), 1 } };
          u = mul_m55_v5(T1, &u);
          for (Z d = 0; d < 5; ++d)
          {
            u.x[d] /= u.x[4];
          }
          u = mul_m55_v5(T2, &u);
          for (Z d = 0; d < 5; ++d)
          {
            u.x[d] /= u.x[3];
          }
          C w = u.x[0] + I * u.x[1];
          R depth = (u.x[2] + 1) / 2; // near 0, far 1
          R a = -log(depth);
          if (a > 0 && depth > 0)
          {
            C pz0 = (WIDTH / 2.0 - 0.5) + I * (HEIGHT / 2.0 - 0.5);
            C pz = w * hypot(WIDTH, HEIGHT) / 4.0 + pz0;
            accum_plot(A, creal(pz), cimag(pz), r, g, b, a);
            t = t + 1;
          }
          z = z * z + c;
        }
      }
    }
  }
  return t;
}
#endif
```

```
int wucleus(C *z0, C c, N period) {
  R eps = nextafter(16, 17) - 16;
  R er2 = 16;
  C z = *z0;
  for (N j = 0; j < 256; ++j) {
    C dz = 1.0;
    for (N k = 0; k < period; ++k) {
      dz = 2.0 * dz * z;
      z = z * z + c;
    }
    R z2 = cabs(z);
    if (! (z2 < er2)) {
      break;
    }
    z = *z0 - (z - *z0) / (dz - 1.0);
    R e = cabs(z - *z0);
    *z0 = z;
    if (e <= eps) {
      return 1;
    }
  }
  return 0;
}

R interior_distance(C *w, C c, N period, R pixel_size) {
  if (wucleus(w, c, period)) {
    C z = *w;
    C dz = 1.0;
    C dzdz = 0.0;
    C dc = 0.0;
    C dcdz = 0.0;
    for (N j = 0; j < period; ++j) {
      dcdz = 2.0 * (z * dcdz + dz * dc);
      dc = 2.0 * z * dc + 1.0;
      dzdz = 2.0 * (dz * dz + z * dzdz);
      dz = 2.0 * z * dz;
      z = z * z + c;
    }
    return (1.0 - cnorm(dz)) / (cabs(dcdz + dzdz * dc / (1.0 - dz)) * pixel_size↙
        ↳ );
  }
  return -1.0;
}

R interior_distance_sign(C *w, C c, N period) {
  if (wucleus(w, c, period)) {
    C z = *w;
    C dz = 1.0;
    for (N j = 0; j < period; ++j) {
      dz = 2.0 * z * dz;
      z = z * z + c;
    }
    return 1.0 - cnorm(dz);
  }
  return -1.0;
}
```

```
static ZZ cloud_calculate_interior(cloud *K, N depth, Z i, Z j, C z, P p)
{
  C c = cloud_c(i, j, depth);
  wucleus(&z, c, p);
  if (depth == DIM)
  {
    K->z[j][i] = z;
    K->p[j][i] = p;
    return 1;
  }
  ZZ t = 0;
  for (Z jj = 2 * j; jj < 2 * j + 2; ++ jj)
    for (Z ii = 2 * i; ii < 2 * i + 2; ++ ii)
      t += cloud_calculate_interior(K, depth + 1, ii, jj, z, p);
  return t;
}

static ZZ cloud_calculate_unknown(cloud *K, N depth, Z i, Z j)
{
  C c = cloud_c(i, j, depth);
  C z = 0;
  C dz = 0;
  R mz2 = 1.0/0.0;
  ZZ t = 0;
  R grid_spacing = (4.0 / (1 << depth));
  if (depth == DIM)
  {
    for (P p = 1; p < MAXPERIOD; ++p)
    {
      dz = 2 * z * dz + 1;
      z = z * z + c;
      R z2 = cnorm(z);
      if (! (z2 < 65536))
      {
        break;
      }
      if (z2 < mz2)
      {
        mz2 = z2;
        C z1 = z;
        R de = interior_distance(&z1, c, p, grid_spacing);
        if (de > 0)
        {
          K->z[j][i] = z1;
          K->p[j][i] = p;
          t += 1;
          break;
        }
      }
    }
  }
  else
  {
    for (P p = 1; p < MAXPERIOD; ++p)
```

```
    {
      dz = 2 * z * dz + 1;
      z = z * z + c;
      R z2 = cnorm(z);
      if (! (z2 < 65536))
      {
        R de = sqrt(z2) * log(z2) / (cabs(dz) * grid_spacing);
        if (de < 4 * sqrt(2))
        {
          for (Z jj = 2 * j; jj < 2 * j + 2; ++ jj)
            for (Z ii = 2 * i; ii < 2 * i + 2; ++ ii)
              t += cloud_calculate_unknown(K, depth + 1, ii, jj);
        }
        break;
      }
      if (z2 < mz2)
      {
        mz2 = z2;
        C z1 = z;
        R de = interior_distance(&z1, c, p, grid_spacing);
        if (de > 0)
        {
          if (de > 0.25 * sqrt(2))
          {
            for (Z jj = 2 * j; jj < 2 * j + 2; ++ jj)
              for (Z ii = 2 * i; ii < 2 * i + 2; ++ ii)
                t += cloud_calculate_interior(K, depth + 1, ii, jj, z1, p);
          }
          else
          {
            for (Z jj = 2 * j; jj < 2 * j + 2; ++ jj)
              for (Z ii = 2 * i; ii < 2 * i + 2; ++ ii)
                t += cloud_calculate_unknown(K, depth + 1, ii, jj);
          }
          break;
        }
      }
    }
  }
  return t;
}

static ZZ cloud_calculate(cloud *K, N depth)
{
  ZZ t = 0;
  #pragma omp parallel for schedule(dynamic, 1) reduction(+:t)
  for (Z j = 0; j < 1 << (depth - 1); ++j)
    for (Z i = 0; i < 1 << depth; ++i)
      t += cloud_calculate_unknown(K, depth, i, j);
  return t;
}

static ZZ cloud_count(cloud *K)
{
  ZZ t = 0;
  #pragma omp parallel for schedule(static) reduction(+:t)
  for (Z j = 0; j < 1 << (DIM - 1); ++j)
```

```
    for (Z i = 0; i < 1 << DIM; ++i)
      t += K->p[j][i];
  return t;
}

static void points_from_cloud(points *Q, const cloud *K)
{
  #pragma omp parallel for schedule(static)
  for (Z j = 0; j < 1 << (DIM - 1); ++j)
  {
    for (Z i = 0; i < 1 << DIM; ++i)
    {
      Z p = K->p[j][i];
      if (0 < p && p < MAXPERIOD)
      {
        #pragma omp atomic update
        Q->n[p] += p;
      }
    }
  }
  for (Z p = 1; p < MAXPERIOD; ++p)
  {
    Q->cz[p] = Q->cz[p - 1] + Q->n[p - 1];
  }
  #pragma omp parallel for schedule(static)
  for (Z j = 0; j < 1 << (DIM - 1); ++j)
  {
    for (Z i = 0; i < 1 << DIM; ++i)
    {
      Z p = K->p[j][i];
      if (0 < p && p < MAXPERIOD)
      {
        ZZ k0 = 0;
        #pragma omp atomic capture
        k0 = Q->i[p] += p;
        k0 -= p;
        C c = cloud_c(i, j, DIM);
        C z = K->z[j][i];
        for (ZZ k = k0; k < k0 + p; ++k)
        {
          V4 v = { { creal(c), cimag(c), creal(z), cimag(z) } };
          Q->cz[p][k] = v;
          z = z * z + c;
        }
      }
    }
  }
}

typedef struct
{
  GLFWwindow *window;
  GLuint vbo[2];
  GLuint vao[2];
  GLuint tex[2];
  GLuint fbo;
  GLuint plot;
```

```
  GLint aspect;
  GLint eye;
  GLint shininess;
  GLint colour;
  GLint conjugate;
  GLint period;
  GLint transform1;
  GLint transform1_r;
  GLint transform1_b;
  GLint transform1_rb;
  GLint transform2;
  GLuint flat;
} graphics;

static void graphics_begin_plot(graphics *GL)
{
  glBindFramebuffer(GL_FRAMEBUFFER, GL->fbo);
  glClearColor(0, 0, 0, 0);
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glBindVertexArray(GL->vao[0]);
  glUseProgram(GL->plot);
  glEnable(GL_BLEND);
  glBlendFunc(GL_ONE, GL_ONE);
}

static void graphics_set_transforms(graphics *GL, const M55 *T1, const M55 *T2, ↙
    ↳ const V5 *eye)
{
  // pack
  F eye2[4];
  F transform2[4][4];
  F transform1[4][4], transform1_r[4], transform1_b[4], transform1_rb;
  for (Z i = 0; i < 4; ++i)
  {
    for (int j = 0; j < 4; ++j)
    {
      transform2[i][j] = T2->x[i][j];
      transform1[i][j] = T1->x[i][j];
    }
    transform1_r[i] = T1->x[4][i];
    transform1_b[i] = T1->x[i][4];
    eye2[i] = eye->x[i] / eye->x[4];
  }
  transform1_rb = T1->x[4][4];
  // upload
  glUniformMatrix4fv(GL->transform2, 1, GL_FALSE, &transform2[0][0]);
  glUniformMatrix4fv(GL->transform1, 1, GL_FALSE, &transform1[0][0]);
  glUniform4fv(GL->transform1_r, 1, &transform1_r[0]);
  glUniform4fv(GL->transform1_b, 1, &transform1_b[0]);
  glUniform1f(GL->transform1_rb, transform1_rb);
  glUniform4fv(GL->eye, 1, &eye2[0]);
}

static void graphics_plot_points(graphics *GL, const points *Q)
{
  R phi5 = pow((sqrt(5) + 1) / 2, 5);
  ZZ n = 0;
```

```
  for (Z p = 1; p < MAXPERIOD; ++p)
  {
    if (Q->n[p] > 0)
    {
      R hue = (p - 1) / phi5;
      F r, g, b;
      hsv2rgb(hue, 0.975, 1, &r, &g, &b);
      glUniform3f(GL->colour, r, g, b);
      glUniform1i(GL->period, p);
      for (Z q = 0; q < 2; ++q)
      {
        glUniform1i(GL->conjugate, q);
        glDrawArrays(GL_POINTS, n, Q->n[p]);
      }
      n += Q->n[p];
    }
  }
}

static void graphics_end_plot(graphics *GL)
{
  glDisable(GL_BLEND);
  glBindFramebuffer(GL_FRAMEBUFFER, 0);
  glClearColor(0, 0, 1, 0);
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glBindVertexArray(GL->vao[1]);
  glUseProgram(GL->flat);
  glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
}

static const char *plot_vert =
  "#version 430 core\n"
  "uniform bool conjugate;\n"
  "uniform int period;\n"
  "uniform float aspect;\n"
  "uniform mat4 transform1;\n"
  "uniform vec4 transform1_r;\n"
  "uniform vec4 transform1_b;\n"
  "uniform float transform1_rb;\n"
  "uniform mat4 transform2;\n"
  "layout (location = 0) in vec4 cz;\n"
  "out vec4 position;\n"
  "out vec4 normal;\n"
  "float[5] mul(float[5][5] m, float[5] v)\n"
  "{\n"
  "  float[5] r;\n"
  "  for (int i = 0; i < 5; ++i)\n"
  "  {\n"
  "    float s = 0.0;\n"
  "    for (int j = 0; j < 5; ++j)\n"
  "      s += m[i][j] * v[j];\n"
  "    r[i] = s;\n"
  "  }\n"
  "  return r;\n"
  "}\n"
  "vec2 mul(vec2 a, vec2 b)\n"
  "{\n"
```

```
    "  return vec2(a.x * b.x - a.y * b.y, a.x * b.y + a.y * b.x);\n"
    "}\n"
    "void main(void)\n"
    "{\n"
    // pack transformation
    "  float[5][5] T1, N1;"
    "  for (int i = 0; i < 4; ++i)\n"
    "  {\n"
    "    for (int j = 0; j < 4; ++j)\n"
    "    {\n"
    "      T1[i][j] = transform1[i][j];\n"
    "      N1[i][j] = transform1[i][j];\n"
    "    }\n"
    "    T1[4][i] = transform1_r[i];\n"
    "    T1[i][4] = transform1_b[i];\n"
    "    N1[4][i] = 0.0;\n"
    "    N1[i][4] = 0.0;\n"
    "  }\n"
    "  T1[4][4] = transform1_rb;\n"
    "  N1[4][4] = 1.0;\n"
    "  vec2 c = cz.xy; if (conjugate) c.y = -c.y;\n"
    "  vec2 z = cz.zw; if (conjugate) z.y = -z.y;\n"
    // compute normal
#if 1
    "  vec2 dz = vec2(1.0, 0.0);\n"
    "  vec2 dc = vec2(0.0, 0.0);\n"
    "  for (int p = 0; p < period; ++p)\n"
    "  {\n"
    "    dz = 2.0 * mul(dz, z);\n"
    "    dc = 2.0 * mul(dc, z) + vec2(1.0, 0.0);\n"
    "    z = mul(z, z) + c;\n"
    "  }\n"
    "  float[5] n = float[5](dc.x, dc.y, dz.x - 1.0, dz.y, 0.0);\n"
    "  n = mul(N1, n);\n"
#endif
    // transform
    "  float[5] u = float[5](c.x, c.y, z.x, z.y, 1.0);\n"
    "  u = mul(T1, u);\n"
    "  position = vec4(u[0], u[1], u[2], u[3]);\n"
    "  normal = vec4(n[0], n[1], n[2], n[3]);\n"
    "  vec4 m, v;\n"
    "  for (int i = 0; i < 4; ++i)\n"
    "  {\n"
//  "    m[i] = n[i] / (n[4] != 0.0 ? n[4] : 1.0);\n"
    "    v[i] = u[i] / u[4];\n"
    "  }\n"
    "  if (all(lessThanEqual(abs(v), vec4(2.0))))\n"
    "  {\n"
    "    v[3] = 1.0;\n"
    "    v = transform2 * v;\n"
    "    v.x /= aspect;\n"
//  "  position = v.xyz / v.w;\n"
//  "  normal = m.xyz / (m.w != 0.0 ? m.w : 1.0);\n"
    "    gl_Position = v;\n"
    "  }\n"
    "  else\n"
    "    gl_Position = vec4(0.0/0.0);\n"
```

```
  "}\n"
  ;

static const char *plot_frag =
  "#version 430 core\n"
  "uniform vec3 colour;\n"
  "uniform float shininess;\n"
  "uniform int period;\n"
  "in vec4 position;\n"
  "in vec4 normal;\n"
  "layout (location = 0) out vec4 fcolour;\n"
  "void main(void)\n"
  "{\n"
#if 1
  "  vec4 eye = vec4(0.0, 0.0, 0.0, 0.0);\n"
  "  vec4 normalDir = normalize(normal);\n"
  "  vec4 viewDir = normalize(eye - position);\n"
  "  if (dot(viewDir, normalDir) < 0.0) normalDir = -normalDir;\n"
  "  float diffuse = 0.0;\n"
  "  float specular = 0.0;\n"
  "  for (int x = -2; x <= 2; x += 4)\n"
  "  for (int y = -2; y <= 2; y += 4)\n"
  "  for (int z = -2; z <= 2; z += 4)\n"
  "  for (int w = -2; w <= 2; w += 4)\n"
  "  {\n"
  "    vec4 light = vec4(x, y, z, w) * 2.0;\n"
  "    float distance = length(light - position) + length(eye - position);\n"
  "    float d = 1000.0 / pow(distance, 3.0);\n"
  "    vec4 lightDir = normalize(light - position);\n"
  "    float diffuse1 = max(dot(lightDir, normalDir), 0.0);\n"
  "    if (diffuse1 > 0.0)\n"
  "    {\n"
  "      vec4 halfDir = normalize(lightDir + viewDir);\n"
  "      specular += d * pow(max(dot(halfDir, normalDir), 0.0), shininess);\n"
  "      diffuse += d * diffuse1;\n"
  "    }\n"
  "  }\n"
  "  fcolour = vec4((diffuse * colour + specular * vec3(10.0)) / float(period), ↙
      ↳ 1.0);\n"
#endif
//  "  fcolour = vec4(colour, 1.0);\n"
//  "  gl_FragDepth = depth;\n"
  "}\n"
  ;

static const char *flat_vert =
  "#version 430 core\n"
  "layout (location = 0) in vec4 p;\n"
  "out vec2 coord;\n"
  "void main(void)\n"
  "{\n"
  "  gl_Position = vec4(p.xy, 0.0, 1.0);\n"
  "  coord = p.zw;\n"
  "}\n"
  ;

static const char *flat_frag =
```

```
    "#version 430 core\n"
    "uniform sampler2D tex;\n"
    "in vec2 coord;\n"
    "layout (location = 0) out vec4 fcolour;\n"
    "void main(void)\n"
    "{\n"
    "  vec4 c = texture(tex, coord);\n"
    "  fcolour = vec4(pow(c.rgb / 1024.0, vec3(0.25)), 1.0);\n"
    "}\n"
    ;

  void debug_program(GLuint program, const char *name) {
    if (program) {
      GLint linked = GL_FALSE;
      glGetProgramiv(program, GL_LINK_STATUS, &linked);
      if (linked != GL_TRUE) {
        fprintf(stderr, "%s: OpenGL shader program link failed\n", name);
      }
      GLint length = 0;
      glGetProgramiv(program, GL_INFO_LOG_LENGTH, &length);
      char *buffer = (char *) malloc(length + 1);
      glGetProgramInfoLog(program, length, NULL, buffer);
      buffer[length] = 0;
      if (buffer[0]) {
        fprintf(stderr, "%s: OpenGL shader program info log\n", name);
        fprintf(stderr, "%s\n", buffer);
      }
      free(buffer);
    } else {
      fprintf(stderr, "%s: OpenGL shader program creation failed\n", name);
    }
  }

  void debug_shader(GLuint shader, GLenum type, const char *name) {
    const char *tname = 0;
    switch (type) {
      case GL_VERTEX_SHADER:   tname = "vertex";   break;
      case GL_GEOMETRY_SHADER: tname = "geometry"; break;
      case GL_FRAGMENT_SHADER: tname = "fragment"; break;
      default:                 tname = "unknown";  break;
    }
    if (shader) {
      GLint compiled = GL_FALSE;
      glGetShaderiv(shader, GL_COMPILE_STATUS, &compiled);
      if (compiled != GL_TRUE) {
        fprintf(stderr, "%s: OpenGL %s shader compile failed\n", name, tname);
      }
      GLint length = 0;
      glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &length);
      char *buffer = (char *) malloc(length + 1);
      glGetShaderInfoLog(shader, length, NULL, buffer);
      buffer[length] = 0;
      if (buffer[0]) {
        fprintf(stderr, "%s: OpenGL %s shader info log\n", name, tname);
        fprintf(stderr, "%s\n", buffer);
      }
      free(buffer);
```

```
  } else {
    fprintf(stderr, "%s: OpenGL %s shader creation failed\n", name, tname);
  }
}

void compile_shader(GLint program, GLenum type, const char *name, const GLchar *source) {
  GLuint shader = glCreateShader(type);
  glShaderSource(shader, 1, &source, NULL);
  glCompileShader(shader);
  debug_shader(shader, type, name);
  glAttachShader(program, shader);
  glDeleteShader(shader);
}

GLint compile_program(const char *name, const GLchar *vert, const GLchar *frag) {
  GLint program = glCreateProgram();
  if (vert) { compile_shader(program, GL_VERTEX_SHADER   , name, vert); }
  if (frag) { compile_shader(program, GL_FRAGMENT_SHADER, name, frag); }
  glLinkProgram(program);
  debug_program(program, name);
  return program;
}

void graphics_init(graphics *GL, Z width, Z height, points *Q)
{
  glfwInit();
  glfwWindowHint(GLFW_CLIENT_API, GLFW_OPENGL_API);
  glfwWindowHint(GLFW_CONTEXT_VERSION_MAJOR, 4);
  glfwWindowHint(GLFW_CONTEXT_VERSION_MINOR, 3);
  glfwWindowHint(GLFW_RESIZABLE, GL_FALSE);
  glfwWindowHint(GLFW_DECORATED, GL_FALSE);
  GL->window = glfwCreateWindow(width, height, "Ultimate Anti-Buddhagram", 0, 0);
  glfwMakeContextCurrent(GL->window);
  glewExperimental = GL_TRUE;
  glewInit();
  glGetError(); // discard common error from glew
  glViewport(0, 0, width, height);
  glDisable(GL_DEPTH_TEST);
//  glEnable(GL_PROGRAM_POINT_SIZE);
  glPointSize(1);
  GL->plot = compile_program("plot", plot_vert, plot_frag);
  glUseProgram(GL->plot);
  GL->transform1 = glGetUniformLocation(GL->plot, "transform1");
  GL->transform1_r = glGetUniformLocation(GL->plot, "transform1_r");
  GL->transform1_b = glGetUniformLocation(GL->plot, "transform1_b");
  GL->transform1_rb = glGetUniformLocation(GL->plot, "transform1_rb");
  GL->transform2 = glGetUniformLocation(GL->plot, "transform2");
  GL->conjugate = glGetUniformLocation(GL->plot, "conjugate");
  GL->period = glGetUniformLocation(GL->plot, "period");
  GL->colour = glGetUniformLocation(GL->plot, "colour");
  GL->shininess = glGetUniformLocation(GL->plot, "shininess");
  glUniform1f(GL->shininess, 100);
  GL->eye = glGetUniformLocation(GL->plot, "eye");
  GL->aspect = glGetUniformLocation(GL->plot, "aspect");
```

```
900     glUniform1f(GL->aspect, width / (R) height);
        GL->flat = compile_program("flat", flat_vert, flat_frag);
        glGenTextures(2, &GL->tex[0]);
        glActiveTexture(GL_TEXTURE1);
        glBindTexture(GL_TEXTURE_2D, GL->tex[1]);
905     glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT, width, height, 0, ↙
            ↳ GL_DEPTH_COMPONENT, GL_FLOAT, 0);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
910     glActiveTexture(GL_TEXTURE0);
        glBindTexture(GL_TEXTURE_2D, GL->tex[0]);
        glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA32F, width, height, 0, GL_RGBA, GL_FLOAT↙
            ↳ , 0);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
915     glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
        glGenFramebuffers(1, &GL->fbo);
        glBindFramebuffer(GL_FRAMEBUFFER, GL->fbo);
        glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0, GL_TEXTURE_2D, GL↙
            ↳ ->tex[0], 0);
920     glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D, GL↙
            ↳ ->tex[1], 0);
        glClear(GL_COLOR_BUFFER_BIT);
        glGenVertexArrays(2, &GL->vao[0]);
        glGenBuffers(2, &GL->vbo[0]);
        glBindVertexArray(GL->vao[0]);
925     glBindBuffer(GL_ARRAY_BUFFER, GL->vbo[0]);
        glBufferData(GL_ARRAY_BUFFER, Q->bytes_used, Q->cz[0], GL_STATIC_DRAW);
        glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 0, 0);
        glEnableVertexAttribArray(0);
        glBindVertexArray(GL->vao[1]);
930     glBindBuffer(GL_ARRAY_BUFFER, GL->vbo[1]);
        float vbo_data[16] =
          { -1, -1, 0, 0
          ,  1, -1, 1, 0
          , -1,  1, 0, 1
935       ,  1,  1, 1, 1
          };
        glBufferData(GL_ARRAY_BUFFER, sizeof(vbo_data), vbo_data, GL_STATIC_DRAW);
        glVertexAttribPointer(0, 4, GL_FLOAT, GL_FALSE, 0, 0);
        glEnableVertexAttribArray(0);
940   }


      static void key_handler(GLFWwindow *window, int key, int scancode, int action, ↙
          ↳ int mods) {
        if (action == GLFW_PRESS) {
945       switch (key) {
            case GLFW_KEY_Q:
            case GLFW_KEY_ESCAPE:
              glfwSetWindowShouldClose(window, GL_TRUE);
              break;
950       }
        }
```

```
  (void) scancode;
  (void) mods;
}

extern int main(int argc, char **argv) {
  (void) argc;
  (void) argv;
  size_t bytes = 0;
  cloud *K = cloud_alloc(&bytes);
  if (! K)
    return 1;
  cloud_clear(K);
  ZZ points_c = cloud_calculate(K, DIM);
  ZZ points_z = cloud_count(K);
  fprintf(stderr, "%lld\n%lld\n", points_c, points_z);
#if 0
  accum *A = accum_alloc();
  image *G = image_alloc();
#endif

  R aspect = 1;//(WIDTH / (R) HEIGHT);
  R zoom = 1;
  R near = 2, far = 10, left = -1, right = 1, top = 1, bottom = -1, in = -1, out
    ↘ = 1;
  M55 offset2 =
    { { { 4, 0, 0,  0, 0 }
      , { 0, 4, 0,  0, 0 }
      , { 0, 0, 4, -6, 0 }
      , { 0, 0, 0,  1, 0 }
      , { 0, 0, 0,  0, 1 }
      }
    };
  M55 frustum2 =
    { { { 2 * near * aspect / (zoom * (right - left)), 0, (right + left) / (
      ↘ right - left), 0, 0 }
      , { 0, 2 * near / (zoom * (top - bottom)), (top + bottom) / (top - bottom)
      ↘ , 0, 0 }
      , { 0, 0, -(far + near) / (far - near), -2 * far * near / (far - near), 0
      ↘ }
      , { 0, 0, -1, 0, 0 }
      , { 0, 0, 0, 0, 1 }
      }
    };
  M55 T2 = mul_m55_m55(&frustum2, &offset2);
  struct quaternion pq0[2] = { { { 1, 0, 0, 0 } }, { { 1, 0, 0, 0 } } };
  struct quaternion pq[2] = { { { 1, 0, 0, 0 } }, { { 1, 0, 0, 0 } } };
  struct quaternion target[2];
  random_unit_quaternion(&target[0]);
  random_unit_quaternion(&target[1]);

  points *Q = points_alloc(points_z);
  points_from_cloud(Q, K);
  cloud_free(K, bytes);
  graphics GL;
  graphics_init(&GL, WIDTH, HEIGHT, Q);
  points_free_data(Q);
```

```
#define pipeline 8
  size_t ppmbytes = WIDTH * (HEIGHT * 3);
  GLsync syncs[pipeline];
  GLuint pbo[pipeline];
  glGenBuffers(pipeline, &pbo[0]);
  for (int i = 0; i < pipeline; ++i) {
    glBindBuffer(GL_PIXEL_PACK_BUFFER, pbo[i]);
    glBufferStorage(GL_PIXEL_PACK_BUFFER, ppmbytes, 0, GL_MAP_READ_BIT);
  }

  glfwSetKeyCallback(GL.window, key_handler);
  glfwPollEvents();
  ZZ frame = 0;
  const int speed = 65 * 16;
  while (! glfwWindowShouldClose(GL.window))
  {
    if ((frame + pipeline) / speed >= 8) break;
    GLint e = glGetError();
    if (e) fprintf(stderr, "E: %d\n", e);
    for (Z q = 0; q < 2; ++q)
    {
      if (frame % speed == 0)
      {
        random_unit_quaternion(&target[q]);
      }
      qslerp(&pq[q], &pq[q], &target[q], 1e-3);
    }
    M55 center =
      { { { 1, 0, 0, 0, -v0.x[0] }
        , { 0, 1, 0, 0, -v0.x[1] }
        , { 0, 0, 1, 0, -v0.x[2] }
        , { 0, 0, 0, 1, -v0.x[3] }
        , { 0, 0, 0, 0, 1 }
        }
      };
    struct matrix4 rotate0;
    mquat2(&rotate0, &pq[0], &pq[1]);
    M55 rotate1 =
      { { { 1, 0, 0, 0, 0 }
        , { 0, 1, 0, 0, 0 }
        , { 0, 0, 1, 0, 0 }
        , { 0, 0, 0, 1, 0 }
        , { 0, 0, 0, 0, 1 }
        }
      };
    M55 rotate1T =
      { { { 1, 0, 0, 0, 0 }
        , { 0, 1, 0, 0, 0 }
        , { 0, 0, 1, 0, 0 }
        , { 0, 0, 0, 1, 0 }
        , { 0, 0, 0, 0, 1 }
        }
      };
    Z c = 0;
    for (Z a = 0; a < 4; ++a)
      for (Z b = 0; b < 4; ++b)
      {
```

```
            rotate1 .x[a][b] = rotate0.m[c];
            rotate1T.x[b][a] = rotate0.m[c];
            ++c;
          }
        R focus = (4 * fabs(((frame % speed) + 0.5) / speed - 0.5) - 1) * 2 + 6;
        M55 offset =
          { { { 1, 0, 0, 0,  0 }
            , { 0, 1, 0, 0,  0 }
            , { 0, 0, 1, 0,  0 }
            , { 0, 0, 0, 1, -6 }
            , { 0, 0, 0, 0,  1 }
            }
          };
        R depth = 32.0 / speed;
        near = focus - depth;
        far = focus + depth;
        M55 frustum =
          { { { 2 * near / (zoom * (right - left)), 0, 0, (right + left) / (right - left), 0 }
            , { 0, 2 * near / (zoom * (top - bottom)), 0, (top + bottom) / (top - bottom), 0 }
            , { 0, 0, 2 * near / (zoom * (in - out)), (in + out) / (in - out), 0 }
            , { 0, 0, 0, -(far + near) / (far - near), -2 * far * near / (far - near) ) }
            , { 0, 0, 0, -1, 0 }
            }
          };
        M55 ortho =
          { { { 2 / (right - left), 0, 0, - (right + left) / (right - left) }
            , { 0, 2 / (top - bottom), 0, 0, - (top + bottom) / (top - bottom) }
            , { 0, 0, 2 / (out - in), 0, - (out + in) / (out - in) }
            , { 0, 0, 0, -2 / (far - near), - (far + near) / (far - near) }
            , { 0, 0, 0, 0, 1 }
            }
          };
        M55 m = mul_m55_m55(&rotate1, &center);
        m = mul_m55_m55(&offset, &m);
        m = mul_m55_m55(&ortho, &m);
        V5 eye = { { 0, 0, 0, 6, 1 } };
        eye = mul_m55_v5(&rotate1T, &eye);
        M55 T1 = m;

    #if 0
        accum_clear(A);
        ZZ total = accum_from_cloud(A, K, &T1, &T2);
    //    fprintf(stderr, "%lld\n", total);
        image_from_accum(G, A, total);
    #endif
        graphics_begin_plot(&GL);
        graphics_set_transforms(&GL, &T1, &T2, &eye);
        graphics_plot_points(&GL, Q);
        graphics_end_plot(&GL);
        int k = frame % pipeline;
        glBindBuffer(GL_PIXEL_PACK_BUFFER, pbo[k]);
        GLsync s = glFenceSync(GL_SYNC_GPU_COMMANDS_COMPLETE, 0);
        if (frame - pipeline >= 0) {
```

```
        glWaitSync(syncs[k], 0, GL_TIMEOUT_IGNORED);
        void *buf = glMapBufferRange(GL_PIXEL_PACK_BUFFER, 0, ppmbytes, ↙
            ↳ GL_MAP_READ_BIT);
        fprintf(stdout, "P6\n%d %d\n%d\n", WIDTH, HEIGHT, 255);
        fwrite(buf, ppmbytes, 1, stdout);
        glUnmapBuffer(GL_PIXEL_PACK_BUFFER);
      }
      syncs[k] = s;
      glReadPixels(0, 0, WIDTH, HEIGHT, GL_RGB, GL_UNSIGNED_BYTE, 0);
      glfwSwapBuffers(GL.window);
      frame++;
      glfwPollEvents();
    }
#if 0
    accum_free(A);
    image_free(G);
#endif
    return 0;
}
```

62 ultimate-anti-buddha/UltimateAntiBuddhagram.frag

```
#version 400 compatibility
#info Ultimate Anti-Buddhagram (c) 2019 Claude Heiland-Allen
#info <https://mathr.co.uk/blog/2019-11-20_ultimate_anti-buddhagram.html>

#define providesInit
#define providesColor
#define WANG_HASH
#include "MathUtils.frag"
#include "DE-Raytracer.frag"
#include "Complex.frag"

#group Buddhagram

uniform int MaxPeriod; slider[1,10,1000]
uniform float SliceBase; slider[-2,0,2]
uniform float SliceDepth; slider[0,4,8]
float Slice;
uniform float Thickness; slider[1e-5,1e-2,1] Logarithmic

uniform float Angle1; slider[-1,0,1]
uniform vec3 Rotation1; slider[(-1,-1,-1),(0,0,1),(1,1,1)]
uniform float Angle2; slider[-1,0,1]
uniform vec3 Rotation2; slider[(-1,-1,-1),(0,0,1),(1,1,1)]
uniform vec4 Center; slider[(-2,-2,-2,-2),(0,0,0,0),(2,2,2,2)]
uniform float Zoom; slider[1e-5,1,1e5] Logarithmic

uniform float ColorSpeed; slider[0,0.5,10]

uniform float time;

const float pi = 3.141592653589793;

// quaternion exponential
vec4 qExp(vec4 q)
{
```

```
  float s   = q[0];
  float v0  = q[1];
  float v1  = q[2];
  float v2  = q[3];
  float vn  = sqrt(v0*v0 + v1*v1 + v2*v2);
  float vnc = cos(vn);
  float vns = vn > 0.0 ? sin(vn) / vn : 0.0;
  float se  = exp(s);
  vec4 r;
  r[0] = se * vnc;
  r[1] = v0 * vns;
  r[2] = v1 * vns;
  r[3] = v2 * vns;
  return r;
}

// 4D rotation matrix from two quaternions
mat4 qMatrix(vec4 p, vec4 q)
{
  mat4 m;
  float q0, q1, q2, q3, p0, p1, p2, p3;
  q0 = q[0]; q1 = q[1]; q2 = q[2]; q3 = q[3];
  p0 = p[0]; p1 = p[1]; p2 = p[2]; p3 = p[3];
  m[0][0] =   q0*p0 + q1*p1 + q2*p2 + q3*p3;
  m[0][1] =   q1*p0 - q0*p1 - q3*p2 + q2*p3;
  m[0][2] =   q2*p0 + q3*p1 - q0*p2 - q1*p3;
  m[0][3] =   q3*p0 - q2*p1 + q1*p2 - q0*p3;
  m[1][0] = - q1*p0 + q0*p1 - q3*p2 + q2*p3;
  m[1][1] =   q0*p0 + q1*p1 - q2*p2 - q3*p3;
  m[1][2] = - q3*p0 + q2*p1 + q1*p2 - q0*p3;
  m[1][3] =   q2*p0 + q3*p1 + q0*p2 + q1*p3;
  m[2][0] = - q2*p0 + q3*p1 + q0*p2 - q1*p3;
  m[2][1] =   q3*p0 + q2*p1 + q1*p2 + q0*p3;
  m[2][2] =   q0*p0 - q1*p1 + q2*p2 - q3*p3;
  m[2][3] = - q1*p0 - q0*p1 + q3*p2 + q2*p3;
  m[3][0] = - q3*p0 - q2*p1 + q1*p2 + q0*p3;
  m[3][1] = - q2*p0 + q3*p1 - q0*p2 + q1*p3;
  m[3][2] =   q1*p0 + q0*p1 + q3*p2 + q2*p3;
  m[3][3] =   q0*p0 - q1*p1 - q2*p2 + q3*p3;
  return m;
}

mat4 M;
void init(void)
{
  float phi1 = (sqrt(5.0) - 1.0) / 2.0;
  float random = rand(time + rand(gl_FragCoord.x + rand(gl_FragCoord.y)));
  float dither = mod(phi1 * (float(subframe) + random) + 0.5, 1.0) - 0.5;
  Slice = SliceBase + SliceDepth * dither;
  M = qMatrix
    ( qExp(vec4(0.0, pi * Angle1 * normalize(Rotation1)))
    , qExp(vec4(0.0, pi * Angle2 * normalize(Rotation2)))
    );
}

float DE(vec4 cz)
{
```

```
  vec2 c = cz.xy;
  vec2 z0 = cz.zw;
  vec2 z = z0;
  vec2 dz = vec2(1.0, 0.0);
  float de = 1.0 / 0.0;
  for (int p = 0; p < MaxPeriod; ++p)
  {
    dz = 2.0 * cMul(dz, z);
    z = cSqr(z) + c;
    if (dot(z, z) > 1000.0) break;
    if (dot(dz, dz) > 1000.0) break;
    float de1 = max(length(z - z0), length(dz) - 1.0);
    de = min(de, de1);
  }
  return 0.25 * (de - Thickness);
}

vec3 periodColor(int p)
{
  vec3 ColorBase = vec3(3.0,2.0,1.0) * pi / 3.0;
  vec3 adj = vec3(ColorSpeed * float(p - 1));
  return (vec3(0.5) + 0.5 * cos(ColorBase + adj)) / float(p);
}

vec3 baseColor(vec3 pos, vec3 normal) {
  vec4 cz = vec4(pos, Slice);
  cz *= M;
  cz *= Zoom;
  cz += Center;
  vec2 c = cz.xy;
  vec2 z0 = cz.zw;
  vec2 z = z0;
  vec2 dz = vec2(1.0, 0.0);
  vec4 color = vec4(0.0);
  for (int p = 1; p <= MaxPeriod; ++p)
  {
    dz = 2.0 * cMul(dz, z);
    z = cSqr(z) + c;
    if (dot(z, z) > 1000.0) break;
    if (dot(dz, dz) > 1000.0) break;
    float de1 = max(length(z - z0), length(dz) - 1.0) - Thickness;
    color += clamp(vec4(periodColor(p), 1.0) / (1e-10 + abs(de1)), 0.0, 1e20);
  }
  return color.rgb / color.a * 20.0;
}

float DE(vec3 pos) {
  vec4 cz = vec4(pos, Slice);
  cz *= M;
  cz *= Zoom;
  cz += Center;
  return DE(cz);
}

#preset Mandelbrot
FOV = 0.4
Eye = 0,0,-4
```

```
Target = 0,0,0
Up = 0,1,0
EquiRectangular = false
AutoFocus = false
FocalPlane = 1.025
Aperture = 0
Gamma = 2
ToneMapping = 4
Exposure = 3
Brightness = 5
Contrast = 1
AvgLumin = 0.5,0.5,0.5
Saturation = 1
LumCoeff = 0.212500006,0.715399981,0.0720999986
Hue = 0
GaussianWeight = 1
AntiAliasScale = 2
DepthToAlpha = false
ShowDepth = false
DepthMagnitude = 1
Detail = -5
DetailAO = -4
FudgeFactor = 1
MaxDistance = 1000
MaxRaySteps = 256
Dither = 0.5 Locked
NormalBackStep = 1
AO = 0,0,0,0.64795919
Specular = 0.4
SpecularExp = 41.884817
SpecularMax = 10
SpotLight = 1,1,1,0.400000006
SpotLightDir = 0.100000001,0.34476846
CamLight = 1,1,1,1
CamLightMin = 0
Glow = 1,1,1,0
GlowMax = 20
Fog = 0
HardShadow = 0
ShadowSoft = 2
QualityShadows = false
Reflection = 0
DebugSun = false
BaseColor = 1,1,1
OrbitStrength = 1
X = 0.5,0.600000024,0.600000024,0.699999988
Y = 1,0.600000024,0,0.400000006
Z = 0.800000012,0.779999971,1,0.5
R = 0.400000006,0.699999988,1,0.119999997
BackgroundColor = 0,0,0
GradientBackground = 0.3
CycleColors = false
Cycles = 1.1
EnableFloor = false
FloorNormal = 0,0,1
FloorHeight = 0
FloorColor = 1,1,1
```

```
TrigIter = 5
TrigLimit = 1.10000000000000009
MaxPeriod = 60
SliceDepth = 4
SliceBase = 0
Thick = 0.01
Angle1 = -1
Rotation1 = 0,0,1
Angle2 = 0
Rotation2 = 0,0,1
Center = -0.400000006,0,-0.400000006,0
Zoom = 1
#endpreset


#preset Lobster
FOV = 0.4
Eye = -0.316849924,-2.01670454,1.52978143
Target = -0.23872563,0.857424268,-1.25109615
Up = -0.203022354,-0.678014877,-0.706454352
EquiRectangular = false
Gamma = 2
ToneMapping = 4
Exposure = 3
Brightness = 5
Contrast = 1
Saturation = 1
GaussianWeight = 1
AntiAliasScale = 2
DepthToAlpha = false
ShowDepth = false
DepthMagnitude = 1
Detail = -5
FudgeFactor = 1
NormalBackStep = 1
CamLight = 1,1,1,2
BaseColor = 1,1,1
OrbitStrength = 1
X = 0.5,0.600000024,0.600000024,0.699999988
Y = 1,0.600000024,0,0.400000006
Z = 0.800000012,0.779999971,1,0.5
R = 0.400000006,0.699999988,1,0.119999997
BackgroundColor = 0,0,0
GradientBackground = 0.3
CycleColors = false
Cycles = 1.1
EnableFloor = false
FloorNormal = 0,0,1
FloorHeight = 0
FloorColor = 1,1,1
TrigIter = 5
TrigLimit = 1.10000000000000009
MaxPeriod = 360
SliceDepth = 0.5
SliceBase = -0.95254832
Thick = 0.01
```

```
Angle1 = -0.0648464
Rotation1 = 0,-0.54929578,0.08450706
Angle2 = 0.2013652
Rotation2 = 0,0,1
Center = 0,0,0,0
Zoom = 1
AutoFocus = false
FocalPlane = 1
Aperture = 0
AvgLumin = 0.5,0.5,0.5
LumCoeff = 0.212500006,0.715399981,0.0720999986
Hue = 0
DetailAO = -4
MaxDistance = 1000
MaxRaySteps = 256
Dither = 0.5 Locked
AO = 0,0,0,0.699999988
Specular = 0.4
SpecularExp = 29.62963
SpecularMax = 10
SpotLight = 1,1,1,0.400000006
SpotLightDir = 0.100000001,0.100000001
CamLightMin = 0
Glow = 1,1,1,0
GlowMax = 20
Fog = 0
HardShadow = 0
ShadowSoft = 2
QualityShadows = false
Reflection = 0
DebugSun = false
Angle11:Linear:0:-1:1:1:101:0.3:1:1.7:1:0
#endpreset
```

63 z-to-exp-z-plus-c/Makefile

```
z-to-exp-z-plus-c: z-to-exp-z-plus-c.c
	gcc -std=c99 -Wall -Wextra -pedantic -O3 -fopenmp -march=native -o z-to-↙
	    ↪ exp-z-plus-c z-to-exp-z-plus-c.c -lm

z-to-exp-z-plus-c-henriksen: z-to-exp-z-plus-c-henriksen.c
	gcc -std=c99 -Wall -Wextra -pedantic -O3 -fopenmp -march=native -o z-to-↙
	    ↪ exp-z-plus-c-henriksen z-to-exp-z-plus-c-henriksen.c -lm

z-to-exp-z-plus-c-lyapunov: z-to-exp-z-plus-c-lyapunov.c
	gcc -std=c99 -Wall -Wextra -pedantic -O3 -fopenmp -march=native -o z-to-↙
	    ↪ exp-z-plus-c-lyapunov z-to-exp-z-plus-c-lyapunov.c -lm
```

64 z-to-exp-z-plus-c/z-to-exp-z-plus-c.c

```
// gcc -std=c99 -Wall -Wextra -pedantic -O3 -fopenmp -march=native -o z-to-exp-z↙
    ↪ -plus-c z-to-exp-z-plus-c.c -lm

#include <complex.h>
#include <math.h>
#include <stdbool.h>
```

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

int rand_r(unsigned int *seedp);


typedef float R;
typedef float _Complex C;
#define creal crealf
#define cimag cimagf
#define cabs cabsf
#define cexp cexpf
#define floor floorf

static R cnorm(C z)
{
  return creal(z) * creal(z) + cimag(z) * cimag(z);
}

static R cisnan(C z)
{
  return isnan(creal(z)) || isnan(cimag(z));
}

static bool newton_step(C *dz_out, C *z_out, C z_guess, C c, int period)
{
  C z = z_guess;
  C dz = 1;
  for (int i = 0; i < period; ++i)
  {
    C e = cexp(z);
    dz = e * dz;
    z = e + c;
  }
  *dz_out = dz;
  C z_new = z_guess - (z - z_guess) / (dz - 1);
  *z_out = cisnan(z_new) ? z_guess : z_new;
  return ! cisnan(z_new);
}

static bool newton_steps(C *dz_out, C *z_out, C z_guess, C c, int period, int
  ↪ nsteps)
{
  C dz = 0;
  C z = z_guess;
  for (int i = 0; i < nsteps; ++i)
  {
    bool ok = newton_step(&dz, &z, z, c, period);
    if (!ok) return false;
  }
  *dz_out = dz;
  *z_out = z;
  return true;
}

static bool interior_de(R *de_out, C *dz_out, C z, C c, int p, int steps) {
```

```
  C dz0 = 0;
  C z0 = 0;
  bool ok = newton_steps(&dz0, &z0, z, c, p, steps);
  if (! ok) return false;
  if (cnorm(dz0) <= 1) {
    C z1 = z0;
    C dz1 = 1;
    C dzdz1 = 0;
    C dc1 = 0;
    C dcdz1 = 0;
    for (int j = 0; j < p; ++j) {
      C e = cexp(z1);
      dcdz1 = e * (dcdz1 + dc1 * dz1);
      dc1 = e * dc1 + 1;
      dzdz1 = e * (dzdz1 + dz1 * dz1);
      dz1 = e * dz1;
      z1 = e + c;
    }
    *de_out = (1 - cnorm(dz1)) / cabs(dcdz1 + dzdz1 * dc1 / (1 - dz1));
    *dz_out = dz1;
    return *de_out > 0;
  }
  return false;
}

static void hsv2rgb(R h, R s, R v, R *red, R *grn, R *blu)
{
  h -= floor(h);
  R hue = h * 6;
  R sat = s;
  R bri = v;
        int i = (int) floor(hue);
        R f = hue - i;
        if (! (i & 1))
                f = 1 - f;
        R m = bri * (1 - sat);
        R n = bri * (1 - sat * f);
  R r = 0, g = 0, b = 0;
        switch (i)
        {
        case 6:
        case 0: b = bri;
                g = n;
                r = m;
                break;
        case 1: b = n;
                g = bri;
                r = m;
                break;
        case 2: b = m;
                g = bri;
                r = n;
                break;
        case 3: b = m;
                g = n;
                r = bri;
                break;
```

```
        case 4: b = n;
                g = m;
                r = bri;
                break;
        case 5: b = bri;
                g = m;
                r = n;
    break;
        }
  *red = b;
  *grn = g;
  *blu = r;
}

extern int main(int argc, char **argv)
{
  (void) argc;
  (void) argv;
  srand(time(0));
  int sup = 1;
  int sub = 1;
  int width = 1920 * sup / sub;
  int height = 1080 * sup / sub;
  int bytes = width * height * 3;
  unsigned char *ppm = malloc(bytes);
  int maxiters = 6000;
  C c0 = 0.75 + I * 1.5;
  C r0 = 0.05;
  R phi = (sqrt(5) + 1) / 2;
  R px = r0 * 2 / height;
  #pragma omp parallel for schedule(static, 1)
  for (int j = 0; j < height; ++j)
  {
    unsigned int seed = rand();
    for (int i = 0; i < width; ++i)
    {
      R x = rand_r(&seed) / (R) RAND_MAX - 0.5;
      R y = rand_r(&seed) / (R) RAND_MAX - 0.5;
      int q = (j * width + i) * 3;
      C c = c0 + r0 * (((i + 0.5 + x) / width - 0.5) * width * 2.0 / height + I *
        * ((j + 0.5 + y) / height - 0.5) * 2.0);
      C dz = 0;
      R mz = 1.0 / 0.0;
      int period = 0;
      R de = 0;
      C z1 = c;
      C z2 = c;
      for (int k = 1; k < maxiters; ++k)
      {
        z1 = cexp(z1) + c;
        z2 = cexp(z2) + c;
        z2 = cexp(z2) + c;
        if (cisnan(z2))
        {
          break;
        }
        R d = cnorm(z1 - z2) / cabs(z1 * z2);
```

```
175           if (d < mz)
              {
                mz = d;
                if (interior_de(&de, &dz, z2, c, k, 64))
                {
180               period = k;
                  break;
                }
              }
            }
            R h = (period - 1) * phi / 24;
            R s = period <= 0 ? 0 : 0.5;
            R v = period <= 0 ? 0 : tanh(0.25 * de / px);
            R r, g, b;
            hsv2rgb(h, s, v, &r, &g, &b);
190         ppm[q++] = 255 * r;
            ppm[q++] = 255 * g;
            ppm[q++] = 255 * b;
          }
        }
195     printf("P6\n%d %d\n255\n", width, height);
        fwrite(ppm, bytes, 1, stdout);
        free(ppm);
        return 0;
      }
```

65 z-to-exp-z-plus-c/z-to-exp-z-plus-c.frag

```
#version 130

#include "Progressive2D.frag"
#include "Complex.frag"
5  #info Mandelbrot
#group Mandelbrot

// Number of iterations
uniform int  Iterations; slider[10,200,1000]
10
bool cisnan(vec2 z)
{
  return isnan(z.x) || isnan(z.y);
}
15
bool newton_step(out vec2 dz_out, out vec2 z_out, vec2 z_guess, vec2 c, int ↙
    ↳ period)
{
  vec2 z = z_guess;
  vec2 dz = vec2(1.0, 0.0);
20  for (int i = 0; i < period; ++i)
  {
    vec2 e = cExp(z);
    dz = cMul(e, dz);
    z = e + c;
25  }
  dz_out = dz;
  vec2 z_new = z_guess - cDiv(z - z_guess, dz - vec2(1.0, 0.0));
  z_out = cisnan(z_new) ? z_guess : z_new;
```

```
  return ! cisnan(z_new);
}

bool newton_steps(out vec2 dz_out, out vec2 z_out, vec2 z_guess, vec2 c, int ↙
    ↳ period, int nsteps)
{
  vec2 dz = vec2(0.0, 0.0);
  vec2 z = z_guess;
  for (int i = 0; i < nsteps; ++i)
  {
    bool ok = newton_step(dz, z, z, c, period);
    if (!ok) return false;
  }
  dz_out = dz;
  z_out = z;
  return true;
}

bool interior_de(out float de_out, out vec2 dz_out, vec2 z, vec2 c, int p, int ↙
    ↳ steps) {
  vec2 dz0 = vec2(0.0, 0.0);
  vec2 z0 = vec2(0.0, 0.0);
  bool ok = newton_steps(dz0, z0, z, c, p, steps);
  if (! ok) return false;
  if (cNorm(dz0) <= 1.0) {
    vec2 z1 = z0;
    vec2 dz1 = vec2(1.0, 0.0);
    vec2 dzdz1 = vec2(0.0, 0.0);
    vec2 dc1 = vec2(0.0, 0.0);
    vec2 dcdz1 = vec2(0.0, 0.0);
    for (int j = 0; j < p; ++j) {
      vec2 e = cExp(z1);
      dcdz1 = cMul(e, dcdz1 + cMul(dc1, dz1));
      dc1 = cMul(e, dc1) + vec2(1.0, 0.0);
      dzdz1 = cMul(e, dzdz1 + cMul(dz1, dz1));
      dz1 = cMul(e, dz1);
      z1 = e + c;
    }
    de_out = (1.0 - cNorm(dz1)) / cAbs(dcdz1 + cDiv(cMul(dzdz1, dc1), vec2(1.0, ↙
        ↳ 0.0) - dz1));
    dz_out = dz1;
    return de_out > 0.0;
  }
  return false;
}

void hsv2rgb(float h, float s, float v, out float red, out float grn, out float ↙
    ↳ blu)
{
  h -= floor(h);
  float hue = h * 6.0;
  float sat = s;
  float bri = v;
  int i = int(floor(hue));
  float f = hue - float(i);
  if (! ((i & 1) == 1)) f = 1.0 - f;
  float m = bri * (1.0 - sat);
```

```
  float n = bri * (1.0 - sat * f);
  float r = 0, g = 0, b = 0;
        switch (i)
        {
        case 6:
        case 0: b = bri;
                g = n;
                r = m;
                break;
        case 1: b = n;
                g = bri;
                r = m;
                break;
        case 2: b = m;
                g = bri;
                r = n;
                break;
        case 3: b = m;
                g = n;
                r = bri;
                break;
        case 4: b = n;
                g = m;
                r = bri;
                break;
        case 5: b = bri;
                g = m;
                r = n;
    break;
        }
  red = b;
  grn = g;
  blu = r;
}

vec3 color(vec2 c)
{
  float phi = (sqrt(5.0) + 1.0) / 2.0;
  float px = length(vec4(dFdx(c), dFdy(c)));
  vec2 dz = vec2(0.0, 0.0);
  float mz = 1.0 / 0.0;
  int period = 0;
  float de = 0;
  vec2 z1 = c;
  vec2 z2 = c;
  int k;
  for (k = 1; k < Iterations; ++k)
      {
        z1 = cExp(z1) + c;
        z2 = cExp(z2) + c;
        z2 = cExp(z2) + c;
        if (cisnan(z2)) break;
        float d = cNorm(z1 - z2) / cAbs(cMul(z1, z2));
        if (d < mz)
        {
          mz = d;
          if (interior_de(de, dz, z2, c, k, 64))
```

```
            {
              period = k;
              break;
            }
          }
        }
        float h = float(period - 1) * phi / 24.0;
        float s = period <= 0 ? 0.0 : 0.5;
        float v = period <= 0 ? k == Iterations ? 1.0 : 0.0 : tanh(clamp(de / px, ↙
            ↳ 0.0, 4.0));
        float r, g, b;
        hsv2rgb(h, s, v, r, g, b);
        return vec3(r, g, b);
    }
```

66 z-to-exp-z-plus-c/z-to-exp-z-plus-c-henriksen.c

```
    // gcc -std=c99 -Wall -Wextra -pedantic -O3 -fopenmp -march=native -o z-to-exp-z↙
        ↳ -plus-c z-to-exp-z-plus-c.c -lm

    #include <complex.h>
    #include <math.h>
    #include <stdbool.h>
    #include <stdio.h>
    #include <stdlib.h>
    #include <time.h>

    int rand_r(unsigned int *seedp);

    long double cnorml(long double _Complex z)
    {
      return creall(z) * creall(z) + cimagl(z) * cimagl(z);
    }

    long double cisnanl(long double _Complex z)
    {
      return isnan(creall(z)) || isnan(cimagl(z));
    }

    bool newton_step(long double _Complex *dz_out, long double _Complex *z_out, long↙
        ↳  double _Complex z_guess, long double _Complex c, int period)
    {
      long double _Complex z = z_guess;
      long double _Complex dz = 1;
      for (int i = 0; i < period; ++i)
      {
        long double _Complex e = cexpl(z);
        dz = e * dz;
        z = e + c;
      }
      *dz_out = dz;
      long double _Complex z_new = z_guess - (z - z_guess) / (dz - 1);
      *z_out = cisnanl(z_new) ? z_guess : z_new;
      return ! cisnanl(z_new);
    }

    bool newton_steps(long double _Complex *dz_out, long double _Complex *z_out, ↙
```

```
    ↪ long double _Complex z_guess, long double _Complex c, int period, int ↙
    ↪ nsteps)
{
  long double _Complex dz = 0;
  long double _Complex z = z_guess;
  for (int i = 0; i < nsteps; ++i)
  {
    bool ok = newton_step(&dz, &z, z, c, period);
    if (!ok) return false;
  }
  *dz_out = dz;
  *z_out = z;
  return true;
}

bool interior_de(long double *de_out, long double _Complex *dz_out, long double ↙
    ↪ _Complex z, long double _Complex c, int p, int steps) {
  long double _Complex dz0 = 0;
  long double _Complex z0 = 0;
  bool ok = newton_steps(&dz0, &z0, z, c, p, steps);
  if (! ok) return false;
  if (cnorml(dz0) <= 1) {
    long double _Complex z1 = z0;
    long double _Complex dz1 = 1;
    long double _Complex dzdz1 = 0;
    long double _Complex dc1 = 0;
    long double _Complex dcdz1 = 0;
    for (int j = 0; j < p; ++j) {
      long double _Complex e = cexpl(z1);
      dcdz1 = e * (dcdz1 + dc1 * dz1);
      dc1 = e * dc1 + 1;
      dzdz1 = e * (dzdz1 + dz1 * dz1);
      dz1 = e * dz1;
      z1 = e + c;
    }
    *de_out = (1 - cnorml(dz1)) / cabsl(dcdz1 + dzdz1 * dc1 / (1 - dz1));
    *dz_out = dz1;
    return *de_out > 0;
  }
  return false;
}

void hsv2rgb(long double h, long double s, long double v, long double *red, long↙
    ↪  double *grn, long double *blu)
{
  h -= floor(h);
  long double hue = h * 6;
  long double sat = s;
  long double bri = v;
        int i = (int) floorl(hue);
        long double f = hue - i;
        if (! (i & 1))
                f = 1 - f;
        long double m = bri * (1 - sat);
        long double n = bri * (1 - sat * f);
  long double r = 0, g = 0, b = 0;
        switch (i)
```

```
        {
        case 6:
        case 0: b = bri;
                g = n;
                r = m;
                break;
        case 1: b = n;
                g = bri;
                r = m;
                break;
        case 2: b = m;
                g = bri;
                r = n;
                break;
        case 3: b = m;
                g = n;
                r = bri;
                break;
        case 4: b = n;
                g = m;
                r = bri;
                break;
        case 5: b = bri;
                g = m;
                r = n;
    break;
        }
  *red = b;
  *grn = g;
  *blu = r;
}

int main(int argc, char **argv)
{
  (void) argc;
  (void) argv;
  srand(time(0));
  int sup = 1;
  int sub = 1;
  int width = 1920 * sup / sub;
  int height = 1080 * sup / sub;
  int bytes = width * height * 3;
  unsigned char *ppm = malloc(bytes);
  int maxiters = 100;
  long double _Complex c0 = 0.75 + I * 1.5;
  long double r0 = 0.05;
  long double px = r0 * 2 / height;
  #pragma omp parallel for
  for (int j = 0; j < height; ++j)
  {
    unsigned int seed = rand();
    for (int i = 0; i < width; ++i)
    {
      long double x = rand_r(&seed) / (long double) RAND_MAX - 0.5;
      long double y = rand_r(&seed) / (long double) RAND_MAX - 0.5;
      int q = (j * width + i) * 3;
      long double _Complex c = c0 + r0 * (((i + 0.5 + x) / width - 0.5) * width ↙
```

```
          ↪ * 2.0 / height + I * ((j + 0.5 + y) / height - 0.5) * 2.0);
        long double _Complex z = c;
        long double _Complex dc = 1;
        long double _Complex dz = 1;
        long double l = 0;
        int k = 0;
        for (k = 0; k < maxiters; ++k)
        {
          long double _Complex e = cexpl(z);
          l += logl(cnorml(e));
          dz = e * dz;
                dc = e * dc + 1;
          z = e + c;
          if (cnorml(dz) < 1.0e-6L) { k = -1; break; }
          if (cnorml(z + c) < cnorml(px * 0.1 * (dc + 1))) break;
        }
        l *= 0.5L / (k + 1);
        long double h = 0;
        long double s = 0;
        long double v = l < 0;
        long double r, g, b;
        hsv2rgb(h, s, v, &r, &g, &b);
        ppm[q++] = 255 * r;
        ppm[q++] = 255 * g;
        ppm[q++] = 255 * b;
      }
    }
    printf("P6\n%d %d\n255\n", width, height);
    fwrite(ppm, bytes, 1, stdout);
    free(ppm);
    return 0;
}
```

67 z-to-exp-z-plus-c/z-to-exp-z-plus-c-lyapunov.c

```
// gcc -std=c99 -Wall -Wextra -pedantic -O3 -fopenmp -march=native -o z-to-exp-z↙
    ↪ -plus-c z-to-exp-z-plus-c.c -lm

#include <complex.h>
#include <math.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int rand_r(unsigned int *seedp);

long double cnorml(long double _Complex z)
{
  return creall(z) * creall(z) + cimagl(z) * cimagl(z);
}

long double cisnanl(long double _Complex z)
{
  return isnan(creall(z)) || isnan(cimagl(z));
}
```

```
bool newton_step(long double _Complex *dz_out, long double _Complex *z_out, long double _Complex z_guess, long double _Complex c, int period)
{
  long double _Complex z = z_guess;
  long double _Complex dz = 1;
  for (int i = 0; i < period; ++i)
  {
    long double _Complex e = cexpl(z);
    dz = e * dz;
    z = e + c;
  }
  *dz_out = dz;
  long double _Complex z_new = z_guess - (z - z_guess) / (dz - 1);
  *z_out = cisnanl(z_new) ? z_guess : z_new;
  return ! cisnanl(z_new);
}

bool newton_steps(long double _Complex *dz_out, long double _Complex *z_out, long double _Complex z_guess, long double _Complex c, int period, int nsteps)
{
  long double _Complex dz = 0;
  long double _Complex z = z_guess;
  for (int i = 0; i < nsteps; ++i)
  {
    bool ok = newton_step(&dz, &z, z, c, period);
    if (!ok) return false;
  }
  *dz_out = dz;
  *z_out = z;
  return true;
}

bool interior_de(long double *de_out, long double _Complex *dz_out, long double _Complex z, long double _Complex c, int p, int steps) {
  long double _Complex dz0 = 0;
  long double _Complex z0 = 0;
  bool ok = newton_steps(&dz0, &z0, z, c, p, steps);
  if (! ok) return false;
  if (cnorml(dz0) <= 1) {
    long double _Complex z1 = z0;
    long double _Complex dz1 = 1;
    long double _Complex dzdz1 = 0;
    long double _Complex dc1 = 0;
    long double _Complex dcdz1 = 0;
    for (int j = 0; j < p; ++j) {
      long double _Complex e = cexpl(z1);
      dcdz1 = e * (dcdz1 + dc1 * dz1);
      dc1 = e * dc1 + 1;
      dzdz1 = e * (dzdz1 + dz1 * dz1);
      dz1 = e * dz1;
      z1 = e + c;
    }
    *de_out = (1 - cnorml(dz1)) / cabsl(dcdz1 + dzdz1 * dc1 / (1 - dz1));
    *dz_out = dz1;
    return *de_out > 0;
  }
```

```
  return false;
}

void hsv2rgb(long double h, long double s, long double v, long double *red, long↙
    ↳ double *grn, long double *blu)
{
  h -= floor(h);
  long double hue = h * 6;
  long double sat = s;
  long double bri = v;
        int i = (int) floorl(hue);
        long double f = hue - i;
        if (! (i & 1))
                f = 1 - f;
        long double m = bri * (1 - sat);
        long double n = bri * (1 - sat * f);
  long double r = 0, g = 0, b = 0;
        switch (i)
        {
        case 6:
        case 0: b = bri;
                g = n;
                r = m;
                break;
        case 1: b = n;
                g = bri;
                r = m;
                break;
        case 2: b = m;
                g = bri;
                r = n;
                break;
        case 3: b = m;
                g = n;
                r = bri;
                break;
        case 4: b = n;
                g = m;
                r = bri;
                break;
        case 5: b = bri;
                g = m;
                r = n;
    break;
        }
  *red = b;
  *grn = g;
  *blu = r;
}

int main(int argc, char **argv)
{
  (void) argc;
  (void) argv;
  srand(time(0));
  int sup = 1;
  int sub = 1;
```

```
  int width = 1920 * sup / sub;
  int height = 1080 * sup / sub;
  int bytes = width * height * 3;
  unsigned char *ppm = malloc(bytes);
  int maxiters = 600;
  long double _Complex c0 = 0.75 + I * 1.5;
  long double _Complex r0 = 0.05;
  long double phi = (sqrtl(5) + 1) / 2;
  #pragma omp parallel for
  for (int j = 0; j < height; ++j)
  {
    unsigned int seed = rand();
    for (int i = 0; i < width; ++i)
    {
      long double x = rand_r(&seed) / (long double) RAND_MAX - 0.5;
      long double y = rand_r(&seed) / (long double) RAND_MAX - 0.5;
      int q = (j * width + i) * 3;
      long double _Complex c = c0 + r0 * (((i + 0.5 + x) / width - 0.5) * width ↙
          ↳ * 2.0 / height + I * ((j + 0.5 + y) / height - 0.5) * 2.0);
      long double _Complex z = c;
      long double l = 0;
      for (int k = 0; k < maxiters; ++k)
      {
        long double _Complex e = cexpl(z);
        l += logl(cabsl(e));
        z = e + c;
      }
      l /= maxiters;
      long double h = (l > 0) * phi;
      long double s = (l > 0) ? 0.25 : 0.75;
      long double v = tanhl(0.1 * fabsl(l));
      long double r, g, b;
      hsv2rgb(h, s, v, &r, &g, &b);
      ppm[q++] = 255 * r;
      ppm[q++] = 255 * g;
      ppm[q++] = 255 * b;
    }
  }
  printf("P6\n%d %d\n255\n", width, height);
  fwrite(ppm, bytes, 1, stdout);
  free(ppm);
  return 0;
}
```