mandelbrot-graphics

Claude Heiland-Allen

2015–2019

Contents

1 c/bin/Makefile

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

PKGCONFIG := PKG_CONFIG_PATH="$(prefix)/lib/pkgconfig" pkg-config
COMPILE := $(CC) -g -std=c99 -Wall -Wextra -pedantic -fPIC -O3 -pipe -MMD `$(PKGCONFIG) --cflags mandelbrot-graphics mandelbrot-numerics mandelbrot-symbolics`
LIBS    := `$(PKGCONFIG) --libs mandelbrot-graphics mandelbrot-numerics mandelbrot-symbolics` -lm -lgmp
OBJECTS := $(patsubst %.c,%.o,$(wildcard *.c))
DEPENDS := $(patsubst %.o,%.d,$(OBJECTS))
EXES    := $(patsubst %.o,%,$(OBJECTS))

all: $(EXES)

clean:
	@echo "CLEAN" ; rm -f $(OBJECTS) $(DEPENDS) $(EXES)

install: $(EXES)
	install -d "$(prefix)/bin"
	install -m 755 -t "$(prefix)/bin" $(EXES)

%: %.o
	@echo "EXE     $@" ; gcc -o $@ $< $(LIBS) || ( echo "ERROR   gcc -o $@ $< $(LIBS)" && false )

%.o: %.c
	@echo "O       $@" ; $(COMPILE) -o $@ -c $< || ( echo "ERROR   $(COMPILE) -o $@ $<" && false )

.SUFFIXES:
.PHONY: all clean install
.SECONDARY: $(OBJECTS)

-include $(DEPENDS)
```

2 c/bin/m-algorithm-9.c

```
#include <complex.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mandelbrot-graphics.h>
#include <mandelbrot-numerics.h>

#define IMAGE_E

void initialize_cs(int m, int n, m_d_transform *t, double _Complex *cs)
{
  #pragma omp parallel for
  for (int j = 0; j < n; ++j)
  {
    for (int i = 0; i < m; ++i)
    {
      double _Complex c = i + I * j;
      double _Complex dc = 1;
      m_d_transform_forward(t, &c, &dc);
      int k = i + j * m;
      cs[k] = c;
    }
  }
}

void step_zs(int mn, char *qs, double _Complex *zs, const double _Complex *cs)
{
  #pragma omp parallel for
  for (int i = 0; i < mn; ++i)
  {
    // load
    double _Complex c = cs[i];
    double _Complex z = zs[i];
    // step
    z = z * z + c;
    // compute quadrant
    char q = 1 << ((creal(z) > 0) | ((cimag(z) > 0) << 1));
    // store
    zs[i] = z;
    qs[i] = q;
  }
}

int scan_for_zeroes(int m, int n, int ip, int *ops, double _Complex *ocs, const char *qs, const double _Complex *zs, const double _Complex *ics)
{
  int o = 0;
  // loop over image interior, to avoid tests in inner 3x3 loop
  #pragma omp parallel for
  for (int j = 1; j < n - 1; ++j)
  {
    for (int i = 1; i < m - 1; ++i)
    {
      // find where 4 quadrants meet in 3x3 region
      char q = 0;
      for (int dj = -1; dj <= 1; ++dj)
      {
```

```
          int jdj = j + dj;
          for (int di = -1; di <= 1; ++di)
          {
60          int idi = i + di;
            int kdk = idi + jdj * m;
            q |= qs[kdk];
          }
        }
65      if (q == 0xF)
        {
          // 4 quadrants meet, check for local minimum at center
          double minmz = 1.0/0.0;
          for (int dj = -1; dj <= 1; ++dj)
70        {
            int jdj = j + dj;
            for (int di = -1; di <= 1; ++di)
            {
              int idi = i + di;
75            int kdk = idi + jdj * m;
              double mz = cabs(zs[kdk]);
              minmz = mz < minmz ? mz : minmz;
            }
          }
80        int k = i + j * m;
          double mz = cabs(zs[k]);
          if (mz <= minmz && minmz < 1.0/0.0)
          {
            // we found a probable zero, output it
85          double _Complex ic = ics[k];
            int out;
            #pragma omp atomic capture
            out = o++;
            ops[out] = ip;
90          ocs[out] = ic;
          }
        }
      }
    }
95  return o;
}

#ifdef IMAGE_E
bool accept_period(m_period_filter_t *data, int p)
100 {
  (void) data;
  return p >= 129 && (p % 4) != 1;
}

105 bool reject_period(m_period_filter_t *data, int p)
{
  (void) data;
  return p < 129;
}
110 #endif

#ifdef IMAGE_F
bool accept_period(m_period_filter_t *data, int p)
```

```
{
  (void) data;
  return p >  129 && (p % 4) == 2;
}

bool reject_period(m_period_filter_t *data, int p)
{
  (void) data;
  return p <= 129 || (p % 4) == 1;
}
#endif

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

  const char *filename = "o.png";

  double _Complex center = -1.940985663823497920192982110559 2e+00 + I * ↙
      ↪ 6.482039615741225554727939743991 8e-04;
#ifdef IMAGE_F
  double radius = 5e-11/2;
#else
  double radius = 5e-9;
#endif

  int maxp = 2000;

  int maxiters = 10000;

  int m = 1920;
  int n = 1080;
  int mn = m * n;

  double minfontsize = 1;
  double maxfontsize = 64;

  double er = 25;

  // colouring
  double phi = (sqrt(5) + 1) / 2;
  double gold = 1 / (phi * phi);
  m_pixel_t white = m_pixel_rgba(1, 1, 1, 1);

  // allocate buffers
  double _Complex *cs = malloc(mn * sizeof(*cs));
  double _Complex *zs = malloc(mn * sizeof(*zs));
  char *qs = malloc(mn * sizeof(*qs));
  double _Complex *ocs = malloc(2 * mn * sizeof(*ocs));
  int *ops = malloc(2 * mn * sizeof(*ops));
  m_image *mask = m_image_new(m, n);
  m_image *img = m_image_new(m, n);

  // set image parameters
  m_period_filter_t filter = { &accept_period, &reject_period };
  m_d_transform *t = m_d_transform_rectangular(m, n, center, radius);
```

```
170     m_d_colour_t *colour = m_d_colour_domain(gold / 5, 0.75, 0.75, &filter);

        // periodicity scan
        initialize_cs(m, n, t, cs);
        memset(zs, 0, mn * sizeof(*zs));
175     int o = 0;
        for (int p = 1; p < maxp && o < mn; ++p)
        {
          step_zs(mn, qs, zs, cs);
          o += scan_for_zeroes(m, n, p, ops + o, ocs + o, qs, zs, cs);
180     }

        // render fractal
        m_d_render_scanline_filtered(img, t, er, maxiters, colour, &filter);
        m_image_dirty(img);
185
        // clear mask
        cairo_t *maskcr = cairo_create(m_image_surface(mask));
        cairo_set_source_rgba(maskcr, 1, 1, 1, 1);
        cairo_paint(maskcr);
190     cairo_set_source_rgba(maskcr, 0, 0, 0, 1);

        // setup for labels
        cairo_t *cr = cairo_create(m_image_surface(img));
        cairo_set_source_rgba(cr, 1, 1, 1, 1);
195     cairo_select_font_face(cr, "LMSans10", CAIRO_FONT_SLANT_NORMAL, ↙
            ↳ CAIRO_FONT_WEIGHT_BOLD);

        // draw labels
        for (int i = 0; i < o; ++i)
        {
200       // atom location to image coordinates
          int p = ops[i];
          double _Complex c = ocs[i];
          double _Complex px = c;
          double _Complex dpx = 1;
205       m_d_transform_reverse(t, &px, &dpx);
          double x = creal(px);
          double y = cimag(px);
          if (m_image_in_bounds(mask, x, y) && m_image_peek(mask, x, y) == white)
          {
210         // no atom already at this location
            // we are the lowest period so far
            // so not a multiple, have true period
            if (m_converged == m_d_nucleus(&c, c, p, 16))
            {
215           // check again, in case nucleus is far from located pixel
              px = c;
              if (accept_period(0, p))
              {
                dpx = m_d_filtered_domain_size(c, p, &filter);
220 //fprintf(stderr, "%d %e\n", p, creal(dpx));
                if (! (creal(dpx) < 1.0/0.0))
                  dpx = 0;
              dpx *= 0.5;
            }
225         else
```

```
        {
          dpx = m_d_domain_size(c, p);
        }
        m_d_transform_reverse(t, &px, &dpx);
        x = creal(px);
        y = cimag(px);
        if (m_image_in_bounds(mask, x, y) && m_image_peek(mask, x, y) == white)
        {
          // calculate font size based on atom domain size
          m_shape shape = m_d_shape_discriminant(m_d_shape_estimate(c, p));
          double fs = (shape == m_cardioid ? 1 : 0.5) * cabs(dpx);
          if (p == 1)
            fs = maxfontsize * 2; // period 1 domain is infinite
          double fsr = 1;
          if (0)//(p % periodmod) != periodneq)
          {
            fs = maxfontsize + 3 * log2(fs);
            fsr = 1.5;
          }
          fs = fmax(fmin(fs, maxfontsize), minfontsize);
          if (fs > minfontsize)
          {
            cairo_set_font_size(cr, fs);
            // draw text centered on point
            char sp[100];
            snprintf(sp, 100, "%d", p);
            cairo_text_extents_t e;
            cairo_text_extents(cr, sp, &e);
            double tx = x - e.x_bearing - e.width / 2.0;
            double ty = y - e.y_bearing - e.height / 2.0;
            cairo_move_to(cr, tx, ty);
            cairo_show_text(cr, sp);
            cairo_fill(cr);
          }
          // update mask
          cairo_arc(maskcr, x, y, fmax(1, fsr * fs), 0, 6.283185307179586);
          cairo_close_path(maskcr);
          cairo_fill(maskcr);
          m_image_flush(mask);
        }
      }
    }
  }

  // save image
  m_image_save_png(img, filename);

  // cleanup
  m_image_delete(img);
  m_image_delete(mask);
  m_d_transform_delete(t);
  m_d_colour_delete(colour);
  free(cs);
  free(zs);
  free(qs);
  free(ocs);
  free(ops);
```

```
  // exit
  return 0;
}
```

3 c/bin/m-average-distance.c

```
#include <stdio.h>
#include <mandelbrot-graphics.h>

int main(int argc, char **argv) {
  (void) argc;
  (void) argv;
  complex double c = -0.75 + I * 0.0;
  double r = 1.5;
  m_pixel_t black = m_pixel_rgba(0, 0, 0, 1);
  m_pixel_t white = m_pixel_rgba(1, 1, 1, 1);
  double er = 600;
  int maxiters = 1000000;
  for (int b = 8; b < 16; ++b)
  {
    int w = 1 << b;
    int h = 1 << b;
    m_image *image = m_image_new(w, h);
    if (image) {
      m_d_transform *transform = m_d_transform_rectangular(w, h, c, r);
      if (transform) {
        complex double c0 = 0;
        complex double dc0 = 1;
        m_d_transform_forward(transform, &c0, &dc0);
        double px = cabs(dc0);
        m_d_colour_t *colour = m_d_colour_minimal(black, black, white);
        if (colour) {
          m_d_render_scanline(image, transform, er, maxiters, colour);
          char filename[100];
          snprintf(filename, 100, "%d.png", b);
          m_image_save_png(image, filename);
          double s = 0;
          double n = 0;
          #pragma omp parallel for reduction(+:s) reduction(+:n) schedule(static
              ↘ , 1)
          for (int y1 = 0; y1 < h; ++y1)
          {
            double s0 = 0;
            double s1 = 0;
            for (int x1 = 0; x1 < w; ++x1)
              if (white != m_image_peek(image, x1, y1))
                for (int y2 = 0; y2 < h; ++y2)
                  for (int x2 = 0; x2 < w; ++x2)
                    if (white != m_image_peek(image, x2, y2))
                    {
                      s0 += 1.0;
                      s1 += hypot(x1 - x2, y1 - y2);
                    }
            s1 *= px;
            n = n + s0;
            s = s + s1;
```

```
            }
            s /= n;
            printf("%d\t%.18f\n", b, s);
            fflush(stdout);
            m_d_colour_delete(colour);
          }
          m_d_transform_delete(transform);
        }
        m_image_delete(image);
      }
    }
    return 0;
}
```

4 c/bin/m-buddhabrot-convergence.c

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

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>

#include <mandelbrot-graphics.h>

bool running = false;

#define class_unknown 0
#define class_exterior 1
#define class_interior 2

struct context {
  uint64_t maxiters;
  uint64_t height;
  uint64_t width;

  double dx;
  double dy;

  double exterior_de_threshold_hi;
  double exterior_de_threshold_lo;
  double exterior_de_decrement;

  double interior_de_threshold_hi;
  double interior_de_threshold_lo;
  double interior_de_decrement;

  uint64_t last_class;
  double last_exterior_de;
  double last_interior_de;
  uint64_t last_interior_period;

  uint64_t i;
  uint64_t j;
```

```
  uint64_t exterior_without_de;
  uint64_t exterior_with_de;
  uint64_t boundary_with_interior_and_de;
  uint64_t interior_with_de;
  uint64_t interior_without_de;

  uint64_t interior;
  uint64_t unknown;
  uint64_t level_set_count[];
};

void get_c(struct context *ctx, double *cx, double *cy) {
  *cx = (ctx->i + ctx->dx) / ctx->width * 4.04 - 2.02;
  *cy = (ctx->j + ctx->dy) / ctx->height * 2.02;
}

void advance_ij(struct context *ctx) {
  ++(ctx->i);
  if (ctx->i == ctx->width) {
    ctx->i = 0;
    ++(ctx->j);
    ctx->last_class = class_unknown;
  }
}

void compute_exterior_pixel_without_de(struct context *ctx) {
  const double er2 = 4;
  uint64_t maxiters = ctx->maxiters;
  double cx, cy;
  get_c(ctx, &cx, &cy);
  double zx = cx, zy = cy, zx2, zy2, z2, zxy;
  uint64_t n = 0;
  do {
    zx2 = zx * zx;
    zy2 = zy * zy;
    z2 = zx2 + zy2;
    if (z2 > er2) { break; }
    zxy = zx * zy;
    zx = zx2 - zy2 + cx;
    zy = 2 * zxy + cy;
    ++n;
  } while (n < maxiters);
  if (n < maxiters) {
    ++(ctx->level_set_count[n]);
    ctx->last_class = class_exterior;
    ctx->last_exterior_de -= ctx->exterior_de_decrement;
  } else {
    ++(ctx->unknown);
    ctx->last_class = class_unknown;
  }
  ++(ctx->exterior_without_de);
}

void compute_exterior_pixel_with_de(struct context *ctx) {
  const double er2_lo = 4;
  const double er2_hi = 65536;
```

```
  uint64_t maxiters = ctx->maxiters;
  double cx, cy;
  get_c(ctx, &cx, &cy);
  double zx = cx, zy = cy, zx2, zy2, z2, zxy, dzx = 1, dzy = 0, dzt;
  uint64_t n = 0;
  uint64_t m = maxiters;
  do {
    zx2 = zx * zx;
    zy2 = zy * zy;
    z2 = zx2 + zy2;
    if (z2 > er2_hi) {
      break;
    }
    if (z2 < er2_lo) {
      m = n;
    }
    dzt = 2 * (zx * dzx - zy * dzy) + 1;
    dzy = 2 * (zx * dzy + zy * dzx);
    dzx = dzt;
    zxy = zx * zy;
    zx = zx2 - zy2 + cx;
    zy = 2 * zxy + cy;
    ++n;
  } while (n < maxiters);
  ++m;
  if (n < maxiters && m < maxiters) {
    ++(ctx->level_set_count[m]);
    ctx->last_class = class_exterior;
    ctx->last_exterior_de = sqrt(z2) * log(z2) / hypot(dzx, dzy);
  } else {
    ++(ctx->unknown);
    ctx->last_class = class_unknown;
  }
  ++(ctx->exterior_with_de);
}

void compute_boundary_pixel_with_interior_checking_and_de(struct context *ctx) {
  const double er2_lo = 4;
  const double er2_hi = 65536;
  uint64_t maxiters = ctx->maxiters;
  double cx, cy;
  get_c(ctx, &cx, &cy);
  double zx = cx, zy = cy, zx2, zy2, z2, zxy, dzx = 1, dzy = 0, dzt, mz2 = 1.0 /↙
      ↳ 0.0;
  double interior_de = -1;
  uint64_t n = 0;
  uint64_t m = maxiters;
  uint64_t result = class_unknown;
  uint64_t period = 0;
  do {
    zx2 = zx * zx;
    zy2 = zy * zy;
    z2 = zx2 + zy2;
    if (z2 > er2_hi) {
      result = class_exterior;
      break;
    }
```

```
155       if (z2 < er2_lo) {
            m = n;
          }
          if (z2 < mz2) {
            mz2 = z2;
160         interior_de = -1;
            complex double dz_unused = 0;
            if (m_d_interior_de(&interior_de, &dz_unused, zx + I * zy, cx + I * cy, n ↙
                ↳ + 1, 64)) {
              result = class_interior;
              period = n + 1;
165           break;
            }
          }
          dzt = 2 * (zx * dzx - zy * dzy) + 1;
          dzy = 2 * (zx * dzy + zy * dzx);
170       dzx = dzt;
          zxy = zx * zy;
          zx = zx2 - zy2 + cx;
          zy = 2 * zxy + cy;
          ++n;
175     } while (n < maxiters);
        ++m;
        if (! (m < maxiters)) {
          result = class_unknown;
        }
180     switch (result) {
        case class_interior:
          ++(ctx->interior);
          ctx->last_interior_de = interior_de;
          ctx->last_interior_period = period;
185       break;
        case class_exterior:
          ++(ctx->level_set_count[m]);
          ctx->last_class = class_exterior;
          ctx->last_exterior_de = sqrt(z2) * log(z2) / hypot(dzx, dzy);
190       break;
        default:
          ++(ctx->unknown);
          ctx->last_class = class_unknown;
          break;
195     }
        ++(ctx->boundary_with_interior_and_de);
      }

      void compute_interior_pixel_without_de(struct context *ctx) {
200     ++(ctx->interior);
        ctx->last_interior_de -= ctx->interior_de_decrement;
        ++(ctx->interior_without_de);
      }

205   void compute_interior_pixel_with_de(struct context *ctx) {
        double cx, cy;
        get_c(ctx, &cx, &cy);
        complex double c = cx + I * cy;
        complex double z = 0;
210     complex double dz_unused;
```

```
  uint64_t period = ctx->last_interior_period;
  for (uint64_t i = 0; i < period; ++i) {
    z = z * z + c;
  }
  double interior_de = -1;
  if (m_d_interior_de(&interior_de, &dz_unused, z, c, period, 64)) {
    ++(ctx->interior);
    ctx->last_class = class_interior;
    ctx->last_interior_de = interior_de;
  } else {
    ++(ctx->unknown);
    ctx->last_class = class_unknown;
  }
  ++(ctx->interior_with_de);
}

int main(int argc, char **argv) {
  if (argc > 1) {
    if (strcmp("init", argv[1]) == 0) {
      if (! (argc > 4)) { return 1; }
      uint64_t maxiters = ((uint64_t) 1) << atoi(argv[2]);
      uint64_t height = ((uint64_t) 1) << atoi(argv[3]);
      uint64_t width = 2 * height;
      double pixel_size = 2.02 / height;

      struct context *ctx = calloc(1, sizeof(*ctx));
      ctx->maxiters = maxiters;
      ctx->height = height;
      ctx->width = width;
      ctx->dx = 0.5;
      ctx->dy = 0.5;
      ctx->exterior_de_threshold_hi = 8 * pixel_size;
      ctx->exterior_de_threshold_lo = 4 * pixel_size;
      ctx->exterior_de_decrement = pixel_size;
      ctx->interior_de_threshold_hi = 8 * pixel_size;
      ctx->interior_de_threshold_lo = 4 * pixel_size;
      ctx->interior_de_decrement = pixel_size;

      FILE *out = fopen(argv[4], "wb");
      if (! out) { return 1; }
      if (1 != fwrite(ctx, sizeof(*ctx), 1, out)) { return 1; }
      uint64_t bytes = sizeof(struct context) + maxiters * sizeof(uint64_t);
      uint64_t zero = 0;
      if (-1 == fseek(out, bytes - sizeof(zero), SEEK_SET)) { return 1; }
      if (1 != fwrite(&zero, sizeof(zero), 1, out)) { return 1; }
      fclose(out);
      return 0;
    }

    if (! (argc > 2)) { return 1; }
    int fd = open(argv[2], O_RDWR);
    uint64_t maxiters = 0;
    read(fd, &maxiters, sizeof(maxiters));
    lseek(fd, 0, SEEK_SET);
    uint64_t bytes = sizeof(struct context) + maxiters * sizeof(uint64_t);
    struct context *ctx = mmap(0, bytes, PROT_READ | PROT_WRITE, MAP_SHARED, fd,↙
        ↳ 0);
```

```
    if (strcmp("result", argv[1]) == 0) {
      if (ctx->j < ctx->height) {
        fprintf(stderr, "INCOMPLETE\n");
      } else {
        printf("# levelset area\n");
        for (uint64_t m = 0; m < ctx->maxiters; ++m) {
          double area = ctx->level_set_count[m] * 4.04 / ctx->width * 2.02 / ctx↙
              ↳ ->height;
          if (area > 0) {
            printf("%lu %.16e\n", m, area);
          }
        }
      }
    } else if (strcmp("compute", argv[1]) == 0) {
      running = true;
      while (running && ctx->j < ctx->height) {
        if (ctx->last_class == class_exterior) {
          if (ctx->last_exterior_de > ctx->exterior_de_threshold_hi) {
            compute_exterior_pixel_without_de(ctx);
          } else if (ctx->last_exterior_de > ctx->exterior_de_threshold_lo) {
            compute_exterior_pixel_with_de(ctx);
          } else {
            compute_boundary_pixel_with_interior_checking_and_de(ctx);
          }
        } else if (ctx->last_class == class_interior) {
          if (ctx->last_interior_de > ctx->interior_de_threshold_hi) {
            compute_interior_pixel_without_de(ctx);
          } else if (ctx->last_interior_de > ctx->interior_de_threshold_lo) {
            compute_interior_pixel_with_de(ctx);
          } else {
            compute_boundary_pixel_with_interior_checking_and_de(ctx);
          }
        } else {
          compute_boundary_pixel_with_interior_checking_and_de(ctx);
        }
        advance_ij(ctx);
      }
      if (! (ctx->j < ctx->height)) {
        printf("DONE\n");
      }
    }

    munmap(ctx, bytes);
    return 0;
  }
}
```

5 c/bin/m-cardioid-warping.c

```
#include <complex.h>
#include <math.h>
#include <stdio.h>
#include <mandelbrot-graphics.h>

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

```
  (void) argc;
  (void) argv;
  int w = 256;
  int h = 256;
  double er = 600;
  double maxiters = 100000;
  m_image *image = m_image_new(w, h);
  m_pixel_t red   = m_pixel_rgba(1, 0, 0, 1);
  m_pixel_t black = m_pixel_rgba(0, 0, 0, 1);
  m_pixel_t white = m_pixel_rgba(1, 1, 1, 1);
  m_d_colour_t *colour = m_d_colour_minimal(red, black, white);
  m_d_transform *rect = m_d_transform_rectangular(w, h, 0, 2);
  m_d_transform *card = m_d_transform_cardioid();
  m_d_transform *uncard = m_d_transform_cardioid();
  m_d_transform_invert(uncard);
  m_d_transform *line = m_d_transform_moebius3(1, -I, -1);
  m_d_transform *unline = m_d_transform_moebius3(1, -I, -1);
  m_d_transform_invert(unline);
  for (int x = 0; x < 256; ++x)
  {
    m_d_transform *translate = m_d_transform_linear(x / 64.0, 1.0);
    m_d_transform *t0 = m_d_transform_compose(rect, uncard);
    m_d_transform *t1 = m_d_transform_compose(t0, unline);
    m_d_transform *t2 = m_d_transform_compose(t1, translate);
    m_d_transform *t3 = m_d_transform_compose(t2, line);
    m_d_transform *t4 = m_d_transform_compose(t3, card);
    m_d_render_scanline(image, t4, er, maxiters, colour);
    char filename[100];
    snprintf(filename, 100, "%03d.png", x);
    m_image_save_png(image, filename);
    m_d_transform_delete(translate);
    m_d_transform_delete(t0);
    m_d_transform_delete(t1);
    m_d_transform_delete(t2);
    m_d_transform_delete(t3);
    m_d_transform_delete(t4);
  }
  m_image_delete(image);
  m_d_colour_delete(colour);
  m_d_transform_delete(rect);
  m_d_transform_delete(line);
  m_d_transform_delete(unline);
  m_d_transform_delete(card);
  m_d_transform_delete(uncard);
  return 0;
}
```

6 c/bin/m-dense-misiurewicz.c

```
#include <complex.h>
#include <math.h>
#include <stdio.h>
#include <gmp.h>
#include <mandelbrot-symbolics.h>
#include <mandelbrot-numerics.h>
#include <mandelbrot-graphics.h>
```

```
int main(int argc, char **argv)
{
  (void) argc;
  (void) argv;
  const double twopi = 6.283185307179586;
  int w = 640;
  int h = 360;
  double er = 600;
  double maxiters = 1000000;
  int sharpness = 4;
  m_image *image = m_image_new(w, h);
  m_pixel_t black = m_pixel_rgba(0, 0, 0, 1);
  m_pixel_t white = m_pixel_rgba(1, 1, 1, 1);
  mpq_t angle;
  mpq_init(angle);
  m_binangle bangle;
  m_binangle_init(&bangle);
  if (image)
  {
    cairo_surface_t *surface = m_image_surface(image);
    cairo_t *cr = cairo_create(surface);
    cairo_select_font_face(cr, "LMSans10", CAIRO_FONT_SLANT_NORMAL, ↙
        ↳ CAIRO_FONT_WEIGHT_BOLD);
    cairo_set_font_size(cr, 24);
    cairo_set_source_rgba(cr, 1, 0, 0, 1);
    m_d_colour_t *colour = m_d_colour_minimal(white, black, white);
    if (colour) {
      for (int period = 2; period < 1 << 11; period <<= 1)
      {
        mpz_set_ui(bangle.pre.bits, 1); bangle.pre.length = period;
        mpz_set_ui(bangle.per.bits, 2); bangle.per.length = period;
        m_binangle_to_rational(angle, &bangle);
        complex double ray = m_d_exray_in_do(angle, sharpness, 8 * sharpness * ↙
            ↳ period, 64);
        complex double mc = 0;
        m_d_misiurewicz(&mc, ray, period, 1, 64);
        m_d_misiurewicz_naive(&mc, mc, period, 1, 64);
        complex double bc = 0, bz = 0;
        m_d_interior(&bz, &bc, 0, 0, cexp(I * twopi / period), 1, 64);
        double r = cabs(mc - bc) / period;
        m_d_transform *transform = m_d_transform_rectangular(w, h, mc, r);
        if (transform)
        {
          m_d_render_scanline(image, transform, er, maxiters, colour);
          m_image_dirty(image);
          cairo_move_to(cr, 24, 24);
          char text[100];
          snprintf(text, 100, "%d", period);
          cairo_show_text(cr, text);
          cairo_fill(cr);
          m_image_flush(image);
          char filename[100];
          snprintf(filename, 100, "%06d.png", period);
          m_image_save_png(image, filename);
          m_d_transform_delete(transform);
        }
      }
```

```
      m_d_colour_delete(colour);
    }
    m_image_delete(image);
  }
  m_binangle_clear(&bangle);
  mpq_clear(angle);
  return 0;
}
```

7 c/bin/m-feigenbaum-zoom.c

```
#include <stdio.h>
#include <mandelbrot-graphics.h>

int main(int argc, char **argv) {
  (void) argc;
  (void) argv;
  const complex double c = -1.401155189093314712;
  const double d = 4.669201609102990671853203821578;
  double r = 8;
  int w = 512;
  int h = 512;
  m_pixel_t red   = m_pixel_rgba(1, 0, 0, 1);
  m_pixel_t black = m_pixel_rgba(0, 0, 0, 1);
  m_pixel_t white = m_pixel_rgba(1, 1, 1, 1);
  double er = 600;
  int maxiters = 100000;
  m_image *image = m_image_new(w, h);
  if (image) {
    m_d_colour_t *colour = m_d_colour_minimal(red, black, white);
    if (colour) {
      for (int frame = 0; frame < 15; ++frame) {
        m_d_transform *rect = m_d_transform_rectangular(w, h, c, r);
        double o = -3.0 / 8.0;
        m_d_transform *move = m_d_transform_linear((w + I * h) * o, 1);
        m_d_transform *transform = m_d_transform_compose(move, rect);
        m_d_render_scanline(image, transform, er, maxiters, colour);
        char filename[100];
        snprintf(filename, 100, "%02d.png", frame);
        m_image_save_png(image, filename);
        m_d_transform_delete(transform);
        m_d_transform_delete(rect);
        m_d_transform_delete(move);
        r /= d;
      }
      m_d_colour_delete(colour);
    }
    m_image_delete(image);
  }
  return 0;
}
```

8 c/bin/m-fibonacci-phi-animation.c

```
#include <stdio.h>
#include <mandelbrot-graphics.h>
```

```
int main(int argc, char **argv) {
  (void) argc;
  (void) argv;
  const double twopi = 6.283185307179586;
  const double phi = 1.618033988749895;
  complex double u = cexp(I * twopi / phi) / 2;
  complex double c = u * (1 - u);
  double f = 1 / (phi * phi);
  f *= f;
  int w = 512;
  int h = 512;
  double r = 4;
  m_pixel_t black = m_pixel_rgba(0, 0, 0, 1);
  m_pixel_t white = m_pixel_rgba(1, 1, 1, 1);
  double er = 600;
  int maxiters = 1000000;
  m_image *image = m_image_new(w, h);
  if (image) {
    m_d_colour_t *colour = m_d_colour_minimal(white, black, white);
    if (colour) {
      for (int i = 0; i < 8; ++i) {
        m_d_transform *transform = m_d_transform_rectangular(w, h, c, r);
        if (transform) {
          m_d_render_scanline(image, transform, er, maxiters, colour);
          char filename[100];
          snprintf(filename, 100, "%03d.png", i);
          m_image_save_png(image, filename);
          m_d_transform_delete(transform);
        }
        r *= f;
      }
      m_d_colour_delete(colour);
    }
    m_image_delete(image);
  }
  return 0;
}
```

9 c/bin/m-fibonacci-phi-zoom.c

```
#include <stdio.h>
#include <mandelbrot-graphics.h>

int main(int argc, char **argv) {
  (void) argc;
  (void) argv;
  const double twopi = 6.283185307179586;
  const double phi = 1.618033988749895;
  complex double u = cexp(I * twopi / phi) / 2;
  complex double c = u * (1 - u);
  double f = 1 / (phi * phi);
  f *= f;
  int w = 512;
  int h = 512;
  double r = 4;
  m_pixel_t black = m_pixel_rgba(0, 0, 0, 1);
```

```
  m_pixel_t white = m_pixel_rgba(1, 1, 1, 1);
  double er = 600;
  int maxiters = 1000000;
  m_image *image = m_image_new(w, h);
  if (image) {
    m_d_colour_t *colour = m_d_colour_minimal(white, black, white);
    if (colour) {
      for (int i = 0; i < 8; ++i) {
        m_d_transform *transform = m_d_transform_rectangular(w, h, c, r);
        if (transform) {
          m_d_render_scanline(image, transform, er, maxiters, colour);
          char filename[100];
          snprintf(filename, 100, "%03d.png", i);
          m_image_save_png(image, filename);
          m_d_transform_delete(transform);
        }
        r *= f;
      }
      m_d_colour_delete(colour);
    }
    m_image_delete(image);
  }
  return 0;
}
```

10 c/bin/m-furcation-rainbow.c

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

#include <mandelbrot-numerics.h>
#include <mandelbrot-graphics.h>

static void usage(const char *progname) {
  fprintf
    ( stderr
    , "usage: %s output.png angle...\n"
    , progname
    );
}

static bool arg_rational(const char *arg, mpq_t x) {
  int ok = mpq_set_str(x, arg, 10);
  mpq_canonicalize(x);
  return ok == 0;
}

static const double twopi = 6.283185307179586;

extern int main(int argc, char **argv) {
  if (! (argc > 2)) {
    usage(argv[0]);
    return 1;
  }
  const int maxsteps = 64;
```

```
  const int sharpness = 256;
  mpq_t q;
  mpq_init(q);
  double dperiod = 1;
  for (int arg = 2; arg < argc; ++arg)
  {
    if (! arg_rational(argv[arg], q)) { mpq_clear(q); return 1; }
    if (! (0 < mpq_cmp_si(q, 0, 1) && mpq_cmp_si(q, 1, 1) < 1)) { mpq_clear(q); ↙
        ↳ return 1; }
    dperiod *= mpz_get_d(mpq_denref(q));
  }
  if (dperiod > INT_MAX) { mpq_clear(q); return 1; }
  int height = 1024;
  int width  = height;
  m_image *img = m_image_new(width, height);
  m_pixel_t black = m_pixel_rgba(0, 0, 0, 1);
  for (int y = 0; y < height; ++y)
  for (int x = 0; x < width; ++x)
    m_image_plot(img, x, y, black);
  int period = 1;
  double _Complex rootc = 0.25;
  double _Complex rootz = 0.5;
  double _Complex rooti = 1;
  for (int arg = 2; arg < argc; ++arg)
  {
    arg_rational(argv[arg], q);
    double t = twopi * mpq_get_d(q);

    double _Complex nucleus;
    m_d_nucleus(&nucleus, rootc, period, maxsteps);
    double _Complex bondi = cexp(I * t);
    double _Complex bondc = 0;
    double _Complex bondz = 0;
    m_d_interior(&bondz, &bondc, 0, nucleus, bondi, period, maxsteps);
    for (int step = 0; step < sharpness; ++step)
    {
      double k = (step + 0.5) / (sharpness);
      double k1 = 1 - k;
      double _Complex c = rootc * k1 + k * bondc;
      double _Complex z = rootz * k1 + k * bondz;
      double _Complex i = rooti * k1 + k * bondi;
      m_d_interior(&z, &c, 0, nucleus, i, period, maxsteps);
      double _Complex w = z;
      for (int att = 0; att < period; ++att)
      {
        m_pixel_t colour = m_pixel_hsva(att / (double) period, 1, 1, 1);
        int x = (2 + creal(w)) * width / 4;
        int y = (2 - cimag(w)) * height / 4;
        m_image_plot(img, x, y, colour);
        w = w * w + c;
      }
    }
    period *= mpz_get_ui(mpq_denref(q));;
    rootc = bondc;
    rootz = bondz;
  }
  m_image_dirty(img);
```

```
  m_image_save_png(img, argv[1]);
  m_image_delete(img);
  mpq_clear(q);
  return 0;
}
```

11 c/bin/m-homunculus.c

```
#include <complex.h>
#include <math.h>
#include <stdio.h>
#include <mandelbrot-graphics.h>

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

int main(int argc, char **argv)
{
  (void) argc;
  (void) argv;
  double _Complex c0 = -1.940985663815178627168439 7e+00 + ↙
      ↪ 6.48203957804514366625984 36e-04 * I;
  double r0 = 5e-9;
  int w = 1920 * 4;
  int h = 1080 * 4;
  double er = 600;
  double maxiters = 100000;
  m_image *image = m_image_new(w, h);
  m_pixel_t red   = m_pixel_rgba(1, 0, 0, 1);
  m_pixel_t black = m_pixel_rgba(0, 0, 0, 1);
  m_pixel_t white = m_pixel_rgba(1, 1, 1, 1);
  m_d_transform *transform = m_d_transform_rectangular(w, h, c0, r0);
  #pragma omp parallel for
  for (int j = 0; j < h; ++j)
  {
    for (int i = 0; i < w; ++i)
    {
      double _Complex z = 0;
      double _Complex c = i + I * j;
      double _Complex dc0 = 1;
      m_d_transform_forward(transform, &c, &dc0);
      int p = 0;
      double _Complex z0 = 0;
      double mz1 = 1.0 / 0.0;
      double _Complex z1 = 0;
      bool e = false;
      double de = -1;
      for (int n = 1; n < maxiters; ++n)
      {
        z = z * z + c;
        double mz = cnorm(z);
        if (mz < mz1)
        {
```

```
            z0 = z1;
            mz1 = mz;
            z1 = z;
            p = n;
          }
          if (mz > er)
          {
            e = true;
            break;
          }
        }
        if (e)
        {
        double _Complex a = z1 / z0;
        double _Complex mu = c, tmp, b;
        m_d_nucleus(&mu, mu, p, 64);
        m_d_interior(&tmp, &b, mu, mu, -1, p, 64);
        b = 0.5 * (mu + b);
        double _Complex s = m_d_size(mu, p);
        double t = m_d_domain_size(mu, p);
        double f = 2 * cabs(s) / cabs(t);
        double g = pow(fmin(fmax(cabs(a), 0), 1), 64);
        double _Complex ch = (f * (c - b) + b) * (1 - g) + g * c;
        double _Complex dch = (f * (1 - g) + g) * dc0;
        z = 0;
        double _Complex dc = 0;
        for (int n = 1; n < maxiters; ++n)
        {
          dc = 2 * dc * z + 1;
          z = z * z + ch;
          double mz = cnorm(z);
          if (mz > er)
          {
            de = 2 * cabs(z) * log(cabs(z)) / cabs(dc * dch);
            break;
          }
        }
        }
        m_pixel_t colour = red;
        if (de >= 0)
        {
          colour = m_pixel_mix(black, white, tanh(de));
        }
        m_image_plot(image, i, j, colour);
      }
    }
    m_image_save_png(image, "m-homunculus.png");
    m_d_transform_delete(transform);
    m_image_delete(image);
    return 0;
}
```

12 c/bin/m-island-zoom.c

```
#include <stdio.h>
#include <mandelbrot-graphics.h>
```

```
int main(int argc, char **argv) {
  (void) argc;
  (void) argv;
  const double _Complex r0 = 1;
  int periods[3] = { 3, 4, 5 };
  double _Complex c1s[3] = { -2, I, -1.5 + I * 0.5 };
  int w = 512;
  int h = 512;
  m_pixel_t red   = m_pixel_rgba(1, 0, 0, 1);
  m_pixel_t black = m_pixel_rgba(0, 0, 0, 1);
  m_pixel_t white = m_pixel_rgba(1, 1, 1, 1);
  double er = 600;
  int maxiters = 1000;
  m_image *image = m_image_new(w, h);
  if (image) {
    m_d_colour_t *colour = m_d_colour_minimal(red, black, white);
    if (colour) {
      for (int k = 0; k < 3; ++k) {
        int period = periods[k];
        double _Complex c1 = c1s[k];
        m_d_nucleus(&c1, c1, period, 64);
        double _Complex r1 = m_d_size(c1, period);
        for (int frame = 0; frame < 50; ++frame) {
          double f = (frame + 0.5) / 50;
          double _Complex r = cpow((r1), f) * cpow((r0), 1 - f);
          double _Complex c = c1 / (1 - r1);
          m_d_transform *rect = m_d_transform_rectangular(w, h, 0, 1);
          m_d_transform *move1 = m_d_transform_linear(- c / 2.25, 1);
          m_d_transform *zoom = m_d_transform_linear(0, r * 2.25);
          m_d_transform *move2 = m_d_transform_linear(c, 1);
          m_d_transform *rm1 = m_d_transform_compose(rect, move1);
          m_d_transform *zm2 = m_d_transform_compose(zoom, move2);
          m_d_transform *transform = m_d_transform_compose(rm1, zm2);
          m_d_render_scanline(image, transform, er, maxiters, colour);
          char filename[100];
          snprintf(filename, 100, "%d-%02d.png", k, frame);
          m_image_save_png(image, filename);
          m_d_transform_delete(transform);
          m_d_transform_delete(zm2);
          m_d_transform_delete(rm1);
          m_d_transform_delete(move2);
          m_d_transform_delete(zoom);
          m_d_transform_delete(move1);
          m_d_transform_delete(rect);
        }
      }
      m_d_colour_delete(colour);
    }
    m_image_delete(image);
  }
  return 0;
}
```

13 c/bin/m-misiurewicz-basins.c

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

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mandelbrot-numerics.h>
#include <mandelbrot-symbolics.h>
#include <mandelbrot-graphics.h>

void trace_ray(cairo_t *cr, m_d_transform *t, double _Complex center, double radius, const char *name, const char *s)
{
  struct m_binangle bangle;
  m_binangle_init(&bangle);
  m_binangle_from_string(&bangle, s);
  mpq_t angle;
  mpq_init(angle);
  m_binangle_to_rational(angle, &bangle);
  m_binangle_clear(&bangle);
  m_d_exray_in *ray = m_d_exray_in_new(angle, 8);
  mpq_clear(angle);
  double _Complex q = 0;
  double _Complex dc = 1;
  for (int i = 0; i < 8192; ++i)
  {
    if (m_stepped != m_d_exray_in_step(ray, 16))
    {
      break;
    }
    double _Complex c = m_d_exray_in_get(ray);
    double r = cabs(c - center) / radius;
    if (r > 0.85)
    {
fprintf(stderr, "%s %.18f %.18f\n", name, creal(q), cimag(q));
      q = c;
    }
    m_d_transform_reverse(t, &c, &dc);
    if (i > 0 && 0 < creal(c) && creal(c) < 1920 && 0 < cimag(c) && cimag(c) < 1080)
      cairo_line_to(cr, creal(c), cimag(c));
    else
      cairo_move_to(cr, creal(c), cimag(c));
  }
  cairo_stroke(cr);
  dc = 1;
fprintf(stderr, "%s %.18f %.18f\n", name, creal(q), cimag(q));
  m_d_transform_reverse(t, &q, &dc);
fprintf(stderr, "%s %f %f\n", name, creal(q), cimag(q));
  cairo_move_to(cr, creal(q), cimag(q));
  cairo_show_text(cr, name);
  cairo_fill(cr);
  m_d_exray_in_delete(ray);
}

void draw_ray(cairo_t *cr, m_d_transform *t, double _Complex center, double radius, const char *name, const char *pre, const char *per)
{
  int len = 4 + strlen(pre) + strlen(per);
  char s[len];
```

```
  s[0] = 0;
  strncat(s, ".", len);
  strncat(s, pre + 2, len);
  s[strlen(s)-1] = 0;
  strncat(s, per + 1, len);
fprintf(stderr, "%s\n", s);
  trace_ray(cr, t, center, radius, name, s);
}

void draw_ray2(cairo_t *cr, m_d_transform *t, double _Complex center, double ↙
    ↳ radius, const char *name, const char *pre, const char *per1, const char *↙
    ↳ per2, const char *per3)
{
  int len = 4 + strlen(pre) + strlen(per1) + strlen(per2) + strlen(per3);
  char s[len];
  s[0] = 0;
  strncat(s, ".", len);
  strncat(s, pre + 2, len);
  s[strlen(s)-1] = 0;
  strncat(s, per1 + 1, len);
  s[strlen(s)-1] = 0;
  strncat(s, per2 + 2, len);
  s[strlen(s)-1] = 0;
  strncat(s, per3 + 2, len);
fprintf(stderr, "%s\n", s);
  trace_ray(cr, t, center, radius, name, s);
}

static inline double cabs2(complex double z) {
  return creal(z) * creal(z) + cimag(z) * cimag(z);
}

extern int main(int argc, char **argv) {
  if (argc != 10) {
    fprintf(stderr, "usage: %s out.png width height creal cimag radius maxiters ↙
        ↳ preperiod period\n", argv[0]);
    return 1;
  }
  int width = atoi(argv[2]);
  int height = atoi(argv[3]);
  double _Complex center = atof(argv[4]) + I * atof(argv[5]);
  double radius = atof(argv[6]);
  int maxiters = atoi(argv[7]);
  int preperiod = atoi(argv[8]);
  int period = atoi(argv[9]);
  const char *filename = argv[1];

  m_image *img = m_image_new(width, height);
  m_d_transform *transform =
    m_d_transform_rectangular(width, height, center, radius);
  #pragma omp parallel for schedule(dynamic, 1)
  for (int j = 0; j < height; ++j) {
    complex double *zs = malloc(maxiters * sizeof(*zs));
    int *ps = malloc(maxiters * sizeof(*ps));
    m_compute_t bias = m_unknown;
    for (int i = 0; i < width; ++i) {
      complex double c = i + I * j;
```

```
        complex double dc0 = 1;
        m_d_transform_forward(transform, &c, &dc0);
        double pixelspacing2 = cabs2(dc0);
        double pixelspacing = sqrt(pixelspacing2);
        double de = 0;
        complex double z = c;
        complex double dc = 1;
        complex double dz = 0;
        double mz2 = 1.0 / 0.0;
        int np = 0;
        if (de == 0) {
          for (int n = 1; n < maxiters; ++n) {
            double z2 = cabs2(z);
            if (z2 < mz2) {
              mz2 = z2;
              int p = n;
              if (bias == m_interior) {
                if (m_d_interior_de(&de, &dz, z, c, p, 64)) {
                  de /= pixelspacing;
                  bias = m_interior;
                  break;
                }
              } else {
                zs[np] = z;
                ps[np] = p;
                np++;
              }
            }
            if (z2 > 65536) {
              de = sqrt(z2 / cabs2(dc * dc0)) * log(z2);
              bias = m_exterior;
              break;
            }
            dc = 2 * z * dc + 1;
            z = z * z + c;
          }
        }
        if (de == 0 && bias != m_interior) {
          for (int n = 0; n < np; ++n) {
            if (m_d_interior_de(&de, &dz, zs[n], c, ps[n], 64)) {
              de /= pixelspacing;
              bias = m_interior;
              break;
            }
          }
        }
        m_d_misiurewicz(&z, c, preperiod, period, 64);
        m_d_misiurewicz_naive(&z, z, preperiod, period, 64);
        double hue = carg(z - center) / 6.283185307179586; hue -= floor(hue);
        double sat = cabs(z - c) < radius / 16 ? 1 : 0.25;
        double val = tanh(de + 1);
        m_image_plot(img, i, j, m_pixel_hsva(hue, sat, val, 1));
      }
      free(zs);
      free(ps);
    }
    m_image_dirty(img);
```

```
    cairo_surface_t *surface = m_image_surface(img);
    cairo_t *cr = cairo_create(surface);
    char *prelo = 0, *prehi = 0, *perlo = 0, *perhi = 0;
    double _Complex nucleus;
    m_d_nucleus(&nucleus, center, preperiod, 64);
    m_d_external_angles(&prelo, &prehi, nucleus, preperiod, 0.9, 64);
    fprintf(stderr, "%.18f %.18f\n%s\n%s\n", creal(nucleus), cimag(nucleus), prelo, prehi);
    m_d_nucleus(&nucleus, center, period, 64);
    m_d_external_angles(&perlo, &perhi, nucleus, period, 0.5, 64);
    fprintf(stderr, "%.18f %.18f\n%s\n%s\n", creal(nucleus), cimag(nucleus), perlo, perhi);
    cairo_set_source_rgba(cr, 0, 0, 0, 1);
    cairo_select_font_face(cr, "LMSans10", CAIRO_FONT_SLANT_NORMAL, CAIRO_FONT_WEIGHT_NORMAL);
    cairo_set_font_size(cr, 16);
    draw_ray(cr, transform, center, radius, ".q(p)", prelo, perlo);
    draw_ray(cr, transform, center, radius, ".q(P)", prelo, perhi);
    draw_ray(cr, transform, center, radius, ".Q(p)", prehi, perlo);
    draw_ray(cr, transform, center, radius, ".Q(P)", prehi, perhi);
    //cairo_set_source_rgba(cr, 1, 0, 0, 1);
    draw_ray2(cr, transform, center, radius, ".q(ppP)", prelo, perlo, perlo, perhi);
    draw_ray2(cr, transform, center, radius, ".q(pPp)", prelo, perlo, perhi, perlo);
    //draw_ray2(cr, transform, prelo, perlo, perhi, perhi);
    //cairo_set_source_rgba(cr, 1, 1, 0, 1);
    draw_ray2(cr, transform, center, radius, ".q(Ppp)", prelo, perhi, perlo, perlo);
    //draw_ray2(cr, transform, prelo, perhi, perlo, perhi);
    //draw_ray2(cr, transform, prelo, perhi, perhi, perlo);
    //cairo_set_source_rgba(cr, 0, 1, 0, 1);
    draw_ray2(cr, transform, center, radius, ".Q(ppP)", prehi, perlo, perlo, perhi);
    draw_ray2(cr, transform, center, radius, ".Q(pPp)", prehi, perlo, perhi, perlo);
    //draw_ray2(cr, transform, prehi, perlo, perhi, perhi);
    //cairo_set_source_rgba(cr, 0, 0, 1, 1);
    draw_ray2(cr, transform, center, radius, ".Q(Ppp)", prehi, perhi, perlo, perlo);
    //draw_ray2(cr, transform, prehi, perhi, perlo, perhi);
    //draw_ray2(cr, transform, prehi, perhi, perhi, perlo);
    m_image_save_png(img, filename);
    m_d_transform_delete(transform);
    m_image_delete(img);
    return 0;
}
```

14 c/bin/m-misiurewicz-domains-2.c

```
/*
gcc -std=c99 -Wall -pedantic -Wextra -O3 -march=native -fopenmp \
    -o m-misiurewicz-domains m-misiurewicz-domains.c \
    `PKG_CONFIG_PATH=/path/to/mandelbrot/opt/lib/pkgconfig \
    pkg-config --cflags --libs mandelbrot-graphics`
*/
#include <complex.h>
```

```c
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mandelbrot-numerics.h>
#include <mandelbrot-graphics.h>

static inline double cabs2(complex double z) {
  return creal(z) * creal(z) + cimag(z) * cimag(z);
}

extern int main(int argc, char **argv) {
  if (argc != 9) {
    fprintf(stderr, "usage: %s out.png width height creal cimag radius maxiters period\n", argv[0]);
    return 1;
  }
  int width = atoi(argv[2]);
  int height = atoi(argv[3]);
  double _Complex center = atof(argv[4]) + I * atof(argv[5]);
  double radius = atof(argv[6]);
  int maxiters = atoi(argv[7]);
  int period = atoi(argv[8]);
  const char *filename = argv[1];

  int *pps = calloc(1, maxiters * sizeof(*pps));
  m_image *img = m_image_new(width, height);
  m_d_transform *transform =
    m_d_transform_rectangular(width, height, center, radius);
  #pragma omp parallel for schedule(dynamic, 1)
  for (int j = 0; j < height; ++j) {
    complex double *zs = malloc(maxiters * sizeof(*zs));
    int *ps = malloc(maxiters * sizeof(*ps));
    m_compute_t bias = m_unknown;
    for (int i = 0; i < width; ++i) {
      complex double c = i + I * j;
      complex double dc0 = 1;
      m_d_transform_forward(transform, &c, &dc0);
      double pixelspacing2 = cabs2(dc0);
      double pixelspacing = sqrt(pixelspacing2);
      double de = 0;
      complex double z = c;
      complex double dc = 1;
      complex double zp = c;
      complex double dcp = 1;
      complex double dz = 0;
      int pp = 0;
      double mp2 = 1.0 / 0.0;
      double mz2 = 1.0 / 0.0;
      int np = 0;
      for (int n = 0; n < period; ++n) {
        double z2 = cabs2(z);
        if (z2 < mz2) {
          mz2 = z2;
          int p = n + 1;
          if (bias == m_interior) {
            if (m_d_interior_de(&de, &dz, z, c, p, 64)) {
```

```
                de /= pixelspacing;
                bias = m_interior;
                break;
              }
            } else {
              zs[np] = z;
              ps[np] = p;
              np++;
            }
          }
          if (z2 > 65536) {
            de = sqrt(z2 / cabs2(dc * dc0)) * log(z2);
            bias = m_exterior;
            break;
          }
          dc = 2 * z * dc + 1;
          z = z * z + c;
        }
        if (de == 0) {
          for (int n = 0; n < maxiters - period; ++n) {
            double z2 = cabs2(z);
            if (z2 < mz2) {
              mz2 = z2;
              int p = n + 1 + period;
              if (bias == m_interior) {
                if (m_d_interior_de(&de, &dz, z, c, p, 64)) {
                  de /= pixelspacing;
                  bias = m_interior;
                  break;
                }
              } else {
                zs[np] = z;
                ps[np] = p;
                np++;
              }
            }
            if (z2 > 65536) {
              de = sqrt(z2 / cabs2(dc * dc0)) * log(z2);
              bias = m_exterior;
              break;
            }
            double p2 = cabs2(z - zp);
            if (p2 < mp2) {
              mp2 = p2;
              pp = n;
            }
            dc = 2 * z * dc + 1;
            z = z * z + c;
            dcp = 2 * zp * dcp + 1;
            zp = zp * zp + c;
          }
        }
        if (de == 0 && bias != m_interior) {
          for (int n = 0; n < np; ++n) {
            if (m_d_interior_de(&de, &dz, zs[n], c, ps[n], 64)) {
              de /= pixelspacing;
              bias = m_interior;
```

```
            break;
          }
        }
      }
      double hue = 0;
      if (pp > 0)
      {
        m_d_misiurewicz(&c, c, pp, period, 16);
        m_d_misiurewicz_naive(&c, c, pp, period, 16);
        z = 0;
        for (int n = 0; n < pp; ++n) z = z * z + c;
        complex double dz = 1;
        for (int n = 0; n < period; ++n)
        {
          dz = 2 * z * dz;
          z = z * z + c;
        }
        hue = fabs(carg(dz)) / log(cabs(dz));
      }
      double sat = bias == m_interior ? 0 : (pp > 0) * (0.25 + 0.75 /
        pow(1.0 + sqrt(mp2 / pixelspacing2) / 1024, 2.0));
      m_image_plot(img, i, j,
        m_pixel_hsva(hue, sat, tanh(de + 0.5), 1));
      #pragma omp atomic
      pps[pp]++;
    }
    free(zs);
    free(ps);
  }
  m_image_dirty(img);
  m_image_save_png(img, filename);
  m_d_transform_delete(transform);
  m_image_delete(img);
  for (int pp = 0; pp < maxiters; ++pp)
    if (pps[pp])
      printf("%d\t%d\n", pps[pp], pp);
  free(pps);
  return 0;
}
```

15 c/bin/m-misiurewicz-domains.c

```
/*
gcc -std=c99 -Wall -pedantic -Wextra -O3 -march=native -fopenmp \
    -o m-misiurewicz-domains m-misiurewicz-domains.c \
    `PKG_CONFIG_PATH=/path/to/mandelbrot/opt/lib/pkgconfig \
    pkg-config --cflags --libs mandelbrot-graphics`
*/
#include <complex.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mandelbrot-numerics.h>
#include <mandelbrot-graphics.h>

static inline double cabs2(complex double z) {
```

```
  return creal(z) * creal(z) + cimag(z) * cimag(z);
}

extern int main(int argc, char **argv) {
  if (argc != 9) {
    fprintf(stderr, "usage: %s out.png width height creal cimag radius maxiters ↙
        ↳ period\n", argv[0]);
    return 1;
  }
  int width = atoi(argv[2]);
  int height = atoi(argv[3]);
  double _Complex center = atof(argv[4]) + I * atof(argv[5]);
  double radius = atof(argv[6]);
  int maxiters = atoi(argv[7]);
  int period = atoi(argv[8]);
  const char *filename = argv[1];

  int *pps = calloc(1, maxiters * sizeof(*pps));
  double hue = 9.016994374947426e-2;
  m_image *img = m_image_new(width, height);
  m_d_transform *transform =
    m_d_transform_rectangular(width, height, center, radius);
  #pragma omp parallel for schedule(dynamic, 1)
  for (int j = 0; j < height; ++j) {
    complex double *zs = malloc(maxiters * sizeof(*zs));
    int *ps = malloc(maxiters * sizeof(*ps));
    m_compute_t bias = m_unknown;
    for (int i = 0; i < width; ++i) {
      complex double c = i + I * j;
      complex double dc0 = 1;
      m_d_transform_forward(transform, &c, &dc0);
      double pixelspacing2 = cabs2(dc0);
      double pixelspacing = sqrt(pixelspacing2);
      double de = 0;
      complex double z = c;
      complex double dc = 1;
      complex double zp = c;
      complex double dcp = 1;
      complex double dz = 0;
      int pp = 0;
      double mp2 = 1.0 / 0.0;
      double mz2 = 1.0 / 0.0;
      int np = 0;
      for (int n = 0; n < period; ++n) {
        double z2 = cabs2(z);
        if (z2 < mz2) {
          mz2 = z2;
          int p = n + 1;
          if (bias == m_interior) {
            if (m_d_interior_de(&de, &dz, z, c, p, 64)) {
              de /= pixelspacing;
              bias = m_interior;
              break;
            }
          } else {
            zs[np] = z;
            ps[np] = p;
```

```
            np++;
          }
        }
        if (z2 > 65536) {
          de = sqrt(z2 / cabs2(dc * dc0)) * log(z2);
          bias = m_exterior;
          break;
        }
        dc = 2 * z * dc + 1;
        z = z * z + c;
      }
      if (de == 0) {
        for (int n = 0; n < maxiters - period; ++n) {
          double z2 = cabs2(z);
          if (z2 < mz2) {
            mz2 = z2;
            int p = n + 1 + period;
            if (bias == m_interior) {
              if (m_d_interior_de(&de, &dz, z, c, p, 64)) {
                de /= pixelspacing;
                bias = m_interior;
                break;
              }
            } else {
              zs[np] = z;
              ps[np] = p;
              np++;
            }
          }
          if (z2 > 65536) {
            de = sqrt(z2 / cabs2(dc * dc0)) * log(z2);
            bias = m_exterior;
            break;
          }
          double p2 = cabs2(z - zp);
          if (p2 < mp2) {
            mp2 = p2;
            pp = n;
          }
          dc = 2 * z * dc + 1;
          z = z * z + c;
          dcp = 2 * zp * dcp + 1;
          zp = zp * zp + c;
        }
      }
      if (de == 0 && bias != m_interior) {
        for (int n = 0; n < np; ++n) {
          if (m_d_interior_de(&de, &dz, zs[n], c, ps[n], 64)) {
            de /= pixelspacing;
            bias = m_interior;
            break;
          }
        }
      }
      double sat = bias == m_interior ? 0 : (pp > 0) * (0.25 + 0.75 /
        pow(1.0 + sqrt(mp2 / pixelspacing2) / 1024, 2.0));
      m_image_plot(img, i, j,
```

```
        m_pixel_hsva((pp - 1) * hue, sat, tanh(de + 0.5), 1));
130     #pragma omp atomic
        pps[pp]++;
      }
      free(zs);
      free(ps);
135   }
    m_image_dirty(img);
    m_image_save_png(img, filename);
    m_d_transform_delete(transform);
    m_image_delete(img);
140 for (int pp = 0; pp < maxiters; ++pp)
    if (pps[pp])
      printf("%d\t%d\n", pps[pp], pp);
    free(pps);
    return 0;
145 }
```

16 c/bin/m-mu-unit.c

```
#include <stdio.h>
#include <mandelbrot-graphics.h>

static inline double cnorm(double _Complex z)
5 {
  double x = creal(z);
  double y = cimag(z);
  return x * x + y * y;
}
10
int main(int argc, char **argv) {
  if (argc != 11) {
    fprintf(stderr, "usage: %s out.png width height creal cimag radius maxiters ↙
        ↳ nreal nimag nperiod\n", argv[0]);
    return 1;
15  }
  const char *filename = argv[1];
  int w = atoi(argv[2]);
  int h = atoi(argv[3]);
  double _Complex c0 = atof(argv[4]) + I * atof(argv[5]);
20  double r = atof(argv[6]);
  int maxiters = atoi(argv[7]);
  double _Complex nucleus = atof(argv[8]) + I * atof(argv[9]);
  int period = atoi(argv[10]);
  m_pixel_t red   = m_pixel_rgba(1, 0, 0, 1);
25  m_pixel_t black = m_pixel_rgba(0, 0, 0, 1);
  m_pixel_t white = m_pixel_rgba(1, 1, 1, 1);
  double er = 25;
  double er2 = er * er;
  int retval = 1;
30  m_image *image = m_image_new(w, h);
  double _Complex dc = 0;
  {
    double _Complex z = 0;
    for (int i = 0; i < period; ++i)
35  {
      dc = 2 * z * dc + 1;
```

```
        z = z * z + nucleus;
      }
    }
    if (image) {
      m_d_transform *transform = m_d_transform_rectangular(w, h, c0, r);
      if (transform) {
        m_image_flush(image);
        #pragma omp parallel for
        for (int y = 0; y < h; ++y)
        {
          for (int x = 0; x < w; ++x)
          {
            double _Complex c = x + I * y;
            double _Complex dz = 1;
            m_d_transform_forward(transform, &c, &dz);
            c -= nucleus;
            c /= dc;
            dz /= dc;
            double _Complex z = 0;
            for (int i = 0; i < maxiters; ++i)
            {
              if (cnorm(z) > er2)
                break;
              dz = 2 * z * dz + 1;
              z = z * z + c;
            }
            m_pixel_t px = red;
            if (cnorm(z) > er2)
            {
              double de = tanh(cabs(z) * log(cabs(z)) / cabs(dz));
              px = m_pixel_mix(black, white, de);
            }
            m_image_plot(image, x, y, px);
          }
        }
        m_image_dirty(image);
        m_image_save_png(image, filename);
        m_d_transform_delete(transform);
        retval = 0;
      }
      m_image_delete(image);
    }
    return retval;
}
```

17 c/bin/m-nucleus-basins.c

```
#include <complex.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mandelbrot-numerics.h>
#include <mandelbrot-graphics.h>

static inline double cabs2(complex double z) {
  return creal(z) * creal(z) + cimag(z) * cimag(z);
```

```
}

extern int main(int argc, char **argv) {
  if (argc != 9) {
    fprintf(stderr, "usage: %s out.png width height creal cimag radius maxiters ↙
        ↳ period\n", argv[0]);
    return 1;
  }
  int width = atoi(argv[2]);
  int height = atoi(argv[3]);
  double _Complex center = atof(argv[4]) + I * atof(argv[5]);
  double radius = atof(argv[6]);
  int maxiters = atoi(argv[7]);
  int period = atoi(argv[8]);
  const char *filename = argv[1];

  m_image *img = m_image_new(width, height);
  m_d_transform *transform =
    m_d_transform_rectangular(width, height, center, radius);
  #pragma omp parallel for schedule(dynamic, 1)
  for (int j = 0; j < height; ++j) {
    complex double *zs = malloc(maxiters * sizeof(*zs));
    int *ps = malloc(maxiters * sizeof(*ps));
    m_compute_t bias = m_unknown;
    for (int i = 0; i < width; ++i) {
      complex double c = i + I * j;
      complex double dc0 = 1;
      m_d_transform_forward(transform, &c, &dc0);
      double pixelspacing2 = cabs2(dc0);
      double pixelspacing = sqrt(pixelspacing2);
      double de = 0;
      complex double z = c;
      complex double dc = 1;
      complex double dz = 0;
      double mz2 = 1.0 / 0.0;
      int np = 0;
      if (de == 0) {
        for (int n = 1; n < maxiters; ++n) {
          double z2 = cabs2(z);
          if (z2 < mz2) {
            mz2 = z2;
            int p = n;
            if (bias == m_interior) {
              if (m_d_interior_de(&de, &dz, z, c, p, 64)) {
                de /= pixelspacing;
                bias = m_interior;
                break;
              }
            } else {
              zs[np] = z;
              ps[np] = p;
              np++;
            }
          }
          if (z2 > 65536) {
            de = sqrt(z2 / cabs2(dc * dc0)) * log(z2);
            bias = m_exterior;
```

```
            break;
          }
          dc = 2 * z * dc + 1;
          z = z * z + c;
        }
      }
      if (de == 0 && bias != m_interior) {
        for (int n = 0; n < np; ++n) {
          if (m_d_interior_de(&de, &dz, zs[n], c, ps[n], 64)) {
            de /= pixelspacing;
            bias = m_interior;
            break;
          }
        }
      }
      m_d_nucleus(&z, c, period, 64);
      double hue = carg(z - center) / 6.283185307179586; hue -= floor(hue);
      double sat = cabs(z - c) < radius / 16 ? 1 : 0.25;
      double val = tanh(de + 1);
      m_image_plot(img, i, j, m_pixel_hsva(hue, sat, val, 1));
    }
    free(zs);
    free(ps);
  }
  m_image_dirty(img);
  m_image_save_png(img, filename);
  m_d_transform_delete(transform);
  m_image_delete(img);
  return 0;
}
```

18 c/bin/m-period-scan.c

```
#include <stdio.h>
#include <mandelbrot-graphics.h>
#include <mandelbrot-numerics.h>

struct atom
{
  double _Complex nucleus;
  double size;
  double domain_size;
  int period;
  m_shape shape;
};

int cmp_atom_size(const void *a, const void *b)
{
  const struct atom *x = a;
  const struct atom *y = b;
  double s = x->size;
  double t = y->size;
  if (s > t) return -1;
  if (s < t) return 1;
  return 0;
}
```

```
25  int cmp_atom_domain_size(const void *a, const void *b)
    {
      const struct atom *x = a;
      const struct atom *y = b;
      double s = x->domain_size;
30    double t = y->domain_size;
      if (s > t) return -1;
      if (s < t) return 1;
      return 0;
    }
35
    int cmp_atom_period(const void *a, const void *b)
    {
      const struct atom *x = a;
      const struct atom *y = b;
40    int s = x->period;
      int t = y->period;
      if (s > t) return 1;
      if (s < t) return -1;
      return 0;
45  }

    int main(int argc, char **argv) {
      if (argc != 14) {
        fprintf(stderr, "usage: %s out.png width height creal cimag radius maxiters ↙
            ↳ \\\n"
50      "  mingridsize minfontsize maxfontsize maxatoms periodmod periodneq\n", argv↙
            ↳ [0]);
        return 1;
      }
      const char *filename = argv[1];
      int w = atoi(argv[2]);
55    int h = atoi(argv[3]);
      double _Complex c = atof(argv[4]) + I * atof(argv[5]);
      double r = atof(argv[6]);
      int maxiters = atoi(argv[7]);
      int mingridsize = atoi(argv[8]);
60    double minfontsize = atof(argv[9]);
      double maxfontsize = atof(argv[10]);
      int maxatoms = atoi(argv[11]);
      int periodmod = atoi(argv[12]);
      int periodneq = atoi(argv[13]);
65    m_pixel_t grey = m_pixel_rgba(0.75, 0.75, 0.75, 1);
      m_pixel_t white = m_pixel_rgba(1, 1, 1, 1);
      double er = 600;
      m_image *image = m_image_new(w, h);
      if (image)
70    {
        m_d_transform *transform = m_d_transform_rectangular(w, h, c, r);
        if (transform) {
          m_d_colour_t *colour = m_d_colour_minimal(white, grey, white);
          if (colour) {
75          // render image
            m_d_render_scanline(image, transform, er, maxiters, colour);
            m_image_dirty(image);
            // scan for periods
            int atoms = 0;
```

```
80          for (int grid = mingridsize << 8; grid >= mingridsize; grid >>= 1)
              for (int y = grid/2; y < h; y += grid)
                for (int x = grid/2; x < w; x += grid)
                  atoms++;
            struct atom *as = calloc(1, atoms * sizeof(*as));
85          atoms = 0;
            for (int grid = mingridsize << 8; grid >= mingridsize; grid >>= 1)
              for (int y = grid/2; y < h; y += grid)
                for (int x = grid/2; x < w; x += grid)
                {
90                double _Complex c0 = x + I * y;
                  double _Complex dc0 = grid;
                  m_d_transform_forward(transform, &c0, &dc0);
                  int p = m_d_box_period_do(c0, 4.0 * cabs(dc0), maxiters);
                  if (p > 0)
95                  if (m_converged == m_d_nucleus(&c0, c0, p, 16))
                    {
                      as[atoms].period = m_d_box_period_do(c0, 0.001 * cabs(dc0), 2 ↙
                          ↳ * p);
                      if (as[atoms].period > 0)
                      {
100                     as[atoms].nucleus = c0;
                        as[atoms].size = cabs(m_d_size(c0, as[atoms].period));
                        as[atoms].domain_size = m_d_domain_size(c0, as[atoms].period↙
                            ↳ );
                        as[atoms].shape = m_d_shape_discriminant(m_d_shape_estimate(↙
                            ↳ c0, as[atoms].period));
                        atoms++;
105                   }
                    }
                }
            qsort(as, atoms, sizeof(*as), cmp_atom_domain_size);
            // prepare deduplication buffer
110         m_image *maskimage = m_image_new(w, h);
            cairo_surface_t *masksurface = m_image_surface(maskimage);
            cairo_t *mask = cairo_create(masksurface);
            cairo_set_source_rgba(mask, 1, 1, 1, 1);
            cairo_paint(mask);
115         cairo_set_source_rgba(mask, 0, 0, 0, 1);
            // prepare image
            cairo_surface_t *surface = m_image_surface(image);
            cairo_t *cr = cairo_create(surface);
            cairo_set_source_rgba(cr, 0.5, 0.25, 0.25, 0.75);
120         cairo_set_operator(cr, CAIRO_OPERATOR_MULTIPLY);
            cairo_select_font_face(cr, "LMSans10", CAIRO_FONT_SLANT_NORMAL, ↙
                ↳ CAIRO_FONT_WEIGHT_BOLD);
            int printed = 0;
            for (int a = 0; printed < maxatoms && a < atoms; ++a)
            {
125           // convert to pixel coordinates
              int p = as[a].period;
              double _Complex c0 = as[a].nucleus;
              double _Complex dc0 = p == 1 ? 1 : as[a].domain_size;
              m_d_transform_reverse(transform, &c0, &dc0);
130           double x = creal(c0);
              double y = cimag(c0);
              // calculate text size
```

```
            double fs = (as[a].shape == m_cardioid ? 1 : 0.5) * cabs(dc0);
            if (periodneq >= 0 && (p % periodmod) != periodneq)
              fs = 8 * log2(fs) + maxfontsize;
            fs = fmax(fs, minfontsize);
            cairo_set_font_size(cr, fs);
            // prevent drawing duplicates
            m_image_flush(maskimage);
            if (! (m_image_in_bounds(maskimage, x, y) && white == m_image_peek(maskimage, x, y)))
              continue;
            cairo_arc(mask, x, y, fs, 0, 6.283185307179586);
            cairo_close_path(mask);
            cairo_fill(mask);
            // draw text centered on point
            char sp[100];
            snprintf(sp, 100, "%d", p);
            cairo_text_extents_t e;
            cairo_text_extents(cr, sp, &e);
            x = x - e.width / 2.0;
            y = y - e.height / 2.0;
            cairo_move_to(cr, x - e.x_bearing, y - e.y_bearing);
            cairo_show_text(cr, sp);
            cairo_fill(cr);
            printed++;
          }
          // cleanup
          cairo_destroy(mask);
          m_image_delete(maskimage);
          cairo_destroy(cr);
          m_image_flush(image);
          m_image_save_png(image, filename);
          m_d_colour_delete(colour);
        }
        m_d_transform_delete(transform);
      }
      m_image_delete(image);
    }
    return 0;
}
```

19 c/bin/m-render.c

```
#include <stdio.h>
#include <mandelbrot-graphics.h>

int main(int argc, char **argv) {
  if (argc != 9) {
    fprintf(stderr, "usage: %s out.png width height creal cimag radius maxiters rgb\n", argv[0]);
    return 1;
  }
  const char *filename = argv[1];
  int w = atoi(argv[2]);
  int h = atoi(argv[3]);
  double _Complex c = atof(argv[4]) + I * atof(argv[5]);
  double r = atof(argv[6]);
  int maxiters = atoi(argv[7]);
```

```
  int rgb = atoi(argv[8]);
  m_pixel_t black = m_pixel_rgba(0, 0, 0, 1);
  m_pixel_t white = m_pixel_rgba(1, 1, 1, 1);
  double phi = (sqrt(5) + 1) / 2;
  double gold = 1 / (phi * phi);
  double er = 25;
  int retval = 1;
  m_image *image = m_image_new(w, h);
  if (image) {
    m_d_transform *transform = m_d_transform_rectangular(w, h, c, r);
    if (transform) {
      m_d_colour_t *colour;
      if (rgb)
        colour = m_d_colour_domain(gold / rgb, 0.5, 1, 0);
      else
        colour = m_d_colour_minimal(white, black, white);
      if (colour) {
        m_d_render_scanline(image, transform, er, maxiters, colour);
        m_image_save_png(image, filename);
        retval = 0;
        m_d_colour_delete(colour);
      }
      m_d_transform_delete(transform);
    }
    m_image_delete(image);
  }
  return retval;
}
```

20 c/bin/m-render-julia.c

```
#include <stdio.h>
#include <mandelbrot-graphics.h>

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

int main(int argc, char **argv) {
  if (argc != 10) {
    fprintf(stderr, "usage: %s out.png width height zreal zimag radius maxiters ↙
        ↳ creal cimag\n", argv[0]);
    return 1;
  }
  const char *filename = argv[1];
  int w = atoi(argv[2]);
  int h = atoi(argv[3]);
  double _Complex z0 = atof(argv[4]) + I * atof(argv[5]);
  double r = atof(argv[6]);
  int maxiters = atoi(argv[7]);
  double _Complex c = atof(argv[8]) + I * atof(argv[9]);
  m_pixel_t red   = m_pixel_rgba(1, 0, 0, 1);
  m_pixel_t black = m_pixel_rgba(0, 0, 0, 1);
  m_pixel_t white = m_pixel_rgba(1, 1, 1, 1);
```

```
  double er = 25;
  double er2 = er * er;
  int retval = 1;
  m_image *image = m_image_new(w, h);
  if (image) {
    m_d_transform *transform = m_d_transform_rectangular(w, h, z0, r);
    if (transform) {
      m_image_flush(image);
      #pragma omp parallel for
      for (int y = 0; y < h; ++y)
      {
        for (int x = 0; x < w; ++x)
        {
          double _Complex z = x + I * y;
          double _Complex dz = 1;
          m_d_transform_forward(transform, &z, &dz);
          for (int i = 0; i < maxiters; ++i)
          {
            if (cnorm(z) > er2)
              break;
            dz = 2 * z * dz;
            z = z * z + c;
          }
          m_pixel_t px = red;
          if (cnorm(z) > er2)
          {
            double de = tanh(cabs(z) * log(cabs(z)) / cabs(dz));
            px = m_pixel_mix(black, white, de);
          }
          m_image_plot(image, x, y, px);
        }
      }
      m_image_dirty(image);
      m_image_save_png(image, filename);
      m_d_transform_delete(transform);
      retval = 0;
    }
    m_image_delete(image);
  }
  return retval;
}
```

21 c/bin/m-stretching-cusps.c

```
#include <mandelbrot-numerics.h>
#include <mandelbrot-graphics.h>
#include <stdio.h>

static const double twopi = 6.283185307179586;

extern int main(int argc, char **argv) {
  int w = 256 * 6;
  int h = 256;
  double r = 2.0/3.0;
  m_pixel_t black = m_pixel_rgba(0, 0, 0, 1);
  m_pixel_t white = m_pixel_rgba(1, 1, 1, 1);
  double er = 100;
```

```
  int maxiters = 8192;
  const char *filename = "out.png";

  if (! (argc == 7)) {
    return 1;
  }
  complex double nucleus = atof(argv[1]) + I * atof(argv[2]);
  int period = atoi(argv[3]);
  mpq_t tzero, tone, tinfinity;
  mpq_init(tzero);
  mpq_init(tone);
  mpq_init(tinfinity);
  mpq_set_str(tzero, argv[4], 10);
  mpq_set_str(tone, argv[5], 10);
  mpq_set_str(tinfinity, argv[6], 10);
  mpq_canonicalize(tzero);
  mpq_canonicalize(tone);
  mpq_canonicalize(tinfinity);

  m_d_nucleus(&nucleus, nucleus, period, 64);
  complex double zero, one, infinity, z;
  m_d_interior(&z, &zero, nucleus, nucleus, cexp(I * twopi * mpq_get_d(tzero)), period, 64);
  m_d_interior(&z, &one, nucleus, nucleus, cexp(I * twopi * mpq_get_d(tone)), period, 64);
  if (mpq_sgn(tinfinity) == 0) {
    m_d_parent(tinfinity, &infinity, &z, nucleus, period, 64);
  } else {
    m_d_interior(&z, &infinity, nucleus, nucleus, cexp(I * twopi * mpq_get_d(tinfinity)), period, 64);
  }

  printf("%.16e %.16e\n", creal(zero), cimag(zero));
  printf("%.16e %.16e\n", creal(one), cimag(one));
  printf("%.16e %.16e\n", creal(infinity), cimag(infinity));

  m_d_transform *rect = m_d_transform_rectangular(w, h, I * 0.99 * r, r);
  m_d_transform *transform = 0;
  switch (m_d_shape(nucleus, period)) {
    case m_cardioid: {
      complex double size = m_d_size(nucleus, period);
      complex double half, cusp;
      m_d_interior(&z, &half, nucleus, nucleus, -1, period, 64);
      m_d_interior(&z, &cusp, nucleus, nucleus, 1, period, 64);
//      complex double size = cusp - half;
      m_d_transform *cardioid = m_d_transform_cardioid();
      m_d_transform *linear = m_d_transform_linear(-nucleus / size, 1 / size);
      m_d_transform *t0 = m_d_transform_compose(cardioid, linear);
      m_d_transform_reverse(t0, &zero, &z);
      m_d_transform_reverse(t0, &one, &z);
      m_d_transform_reverse(t0, &infinity, &z);
      m_d_transform *moebius = m_d_transform_moebius3(zero, one, infinity);
      m_d_transform *t1 = m_d_transform_compose(rect, moebius);
      transform = m_d_transform_compose(t1, t0);
      break;
    }
    case m_circle: {
```

```
      m_d_transform *moebius = m_d_transform_moebius3(zero, one, infinity);
      transform = m_d_transform_compose(rect, moebius);
      break;
    }
    default: {
      return 1;
    }
  }

  m_d_colour_t *colour = m_d_colour_minimal(white, black, white);
  m_image *image = m_image_new(w, h);
  m_d_render_scanline(image, transform, er, maxiters, colour);
  m_image_save_png(image, filename);

  return 0;
}
```

22 c/bin/m-stretching-feigenbaum.c

```
#include <mandelbrot-numerics.h>
#include <mandelbrot-graphics.h>
#include <stdio.h>

extern int main(int argc, char **argv) {
  (void) argc;
  (void) argv;
  int w = 1500;
  int h = 500;
  double r = 2.25;
  m_pixel_t black = m_pixel_rgba(0, 0, 0, 1);
  m_pixel_t white = m_pixel_rgba(1, 1, 1, 1);
  double er = 100;
  int maxiters = 100100;
  const char *filename = "out.png";

  m_d_transform *rect = m_d_transform_rectangular(w, h, -2.5 * r, r);
  m_d_transform *exponential = m_d_transform_exponential(-1.401155189093314712);
  m_d_transform *transform = m_d_transform_compose(rect, exponential);
  m_d_colour_t *colour = m_d_colour_minimal(white, black, white);
  m_image *image = m_image_new(w, h);
  m_d_render_scanline(image, transform, er, maxiters, colour);
  m_image_save_png(image, filename);
  return 0;
}
```

23 c/bin/m-subwake-diagram-a.c

```
#include <mandelbrot-graphics.h>
#include <mandelbrot-numerics.h>
#include <mandelbrot-symbolics.h>
#include <cairo.h>

const double twopi = 6.283185307179586;

void draw_label(m_image *image, m_d_transform *transform, double _Complex c0, ↙
    ↳ const char *text, double pt, m_pixel_t colour) {
```

```
  double _Complex c = c0;
  double _Complex dc = 1;
  m_d_transform_reverse(transform, &c, &dc);
  cairo_surface_t *surface = m_image_surface(image);
  cairo_t *cr = cairo_create(surface);
  cairo_select_font_face(cr, "LMSans10", CAIRO_FONT_SLANT_NORMAL, ↙
    ↳ CAIRO_FONT_WEIGHT_NORMAL);
  cairo_set_font_size(cr, pt);
  cairo_text_extents_t te;
  cairo_text_extents(cr, text, &te);
  cairo_move_to(cr, creal(c) - te.x_bearing - te.width / 2, cimag(c) - te.↙
    ↳ y_bearing - te.height / 2);
  cairo_text_path(cr, text);
  cairo_set_source_rgba(cr, m_pixel_red(colour), m_pixel_green(colour), ↙
    ↳ m_pixel_blue(colour), m_pixel_alpha(colour));
  cairo_fill(cr);
  cairo_destroy(cr);
}

void draw_internal_ray(m_image *image, m_d_transform *transform, int period, ↙
  ↳ double _Complex nucleus, const char *angle, double pt, m_pixel_t colour) {
  int steps = 128;
  mpq_t theta;
  mpq_init(theta);
  mpq_set_str(theta, angle, 10);
  mpq_canonicalize(theta);
  double a = twopi * mpq_get_d(theta);
  mpq_clear(theta);
  double _Complex interior = cos(a) + I * sin(a);

  double _Complex cl = 0, cl2 = 0;
  double _Complex c = nucleus;
  double _Complex z = c;
  cairo_surface_t *surface = m_image_surface(image);
  cairo_t *cr = cairo_create(surface);
  cairo_set_source_rgba(cr, m_pixel_red(colour), m_pixel_green(colour), ↙
    ↳ m_pixel_blue(colour), m_pixel_alpha(colour));
  for (int i = 0; i < steps; ++i) {
    if (2 * i == steps) {
      cl = c;
    }
    if (2 * i == steps + 2) {
      cl2 = c;
    }
    double radius = (i + 0.5) / steps;
    m_d_interior(&z, &c, z, c, radius * interior, period, 64);
    double _Complex pc = c;
    double _Complex pdc = 1;
    m_d_transform_reverse(transform, &pc, &pdc);
    if (i == 0) {
      cairo_move_to(cr, creal(pc), cimag(pc));
    } else {
      cairo_line_to(cr, creal(pc), cimag(pc));
    }
  }
  cairo_stroke(cr);
  if (a != 0) {
```

```
    double t = carg(cl2 - cl);
    cairo_save(cr);
    double _Complex dcl = 1;
    m_d_transform_reverse(transform, &cl, &dcl);
    cairo_translate(cr, creal(cl), cimag(cl));
    cairo_rotate(cr, -t);
    cairo_translate(cr, 0, -pt/3);
    cairo_select_font_face(cr, "LMSans10", CAIRO_FONT_SLANT_NORMAL, ↙
        ↳ CAIRO_FONT_WEIGHT_NORMAL);
    cairo_set_font_size(cr, pt);
    cairo_text_path(cr, angle);
    cairo_fill(cr);
    cairo_restore(cr);
  }
  cairo_destroy(cr);
}

void draw_external_ray(m_image *image, m_d_transform *transform, const char *↙
    ↳ angle, m_pixel_t colour, double dx, double dy) {
  int maxiters = 1024;
  double r = sqrt(2);

  m_binangle btheta;
  m_binangle_init(&btheta);
  m_binangle_from_string(&btheta, angle);
  mpq_t qtheta;
  mpq_init(qtheta);
  m_binangle_to_rational(qtheta, &btheta);
  m_binangle_clear(&btheta);
  m_d_exray_in *ray = m_d_exray_in_new(qtheta, 8);
  mpq_clear(qtheta);

  cairo_surface_t *surface = m_image_surface(image);
  cairo_t *cr = cairo_create(surface);
  cairo_set_source_rgba(cr, m_pixel_red(colour), m_pixel_green(colour), ↙
      ↳ m_pixel_blue(colour), m_pixel_alpha(colour));
  bool first = true;
  for (int i = 0; i < maxiters; ++i) {
    if (m_failed == m_d_exray_in_step(ray, 64)) {
      break;
    }
    double _Complex c = m_d_exray_in_get(ray);
    if (cabs(c + 0.75) > r) {
      continue;
    }
    double t = carg(c + 0.75);
    double _Complex dc = 1;
    m_d_transform_reverse(transform, &c, &dc);
    if (first) {
      cairo_save(cr);
      cairo_translate(cr, creal(c) + dx, cimag(c) + dy);
      cairo_rotate(cr, -t);
      cairo_select_font_face(cr, "LMMono10", CAIRO_FONT_SLANT_NORMAL, ↙
          ↳ CAIRO_FONT_WEIGHT_NORMAL);
      cairo_set_font_size(cr, 48);
      cairo_text_path(cr, angle);
      cairo_fill(cr);
```

```
        cairo_restore(cr);
        cairo_move_to(cr, creal(c) + dx, cimag(c) + dy);
        first = false;
      } else {
        cairo_line_to(cr, creal(c), cimag(c));
      }
    }
    cairo_stroke(cr);
    cairo_destroy(cr);
  }

  int main(int argc, char **argv) {
    (void) argc;
    (void) argv;
    int w = 4096;
    int h = 4096;
    complex double c = -0.75;
    double r = 1.75;
    double er = 600;
    int maxiters = 8192;
    const char *filename = "subwake-diagram-a.png";

    m_pixel_t red   = m_pixel_rgba(1, 0, 0, 1);
    m_pixel_t green = m_pixel_rgba(0, 0.5, 0, 1);
    m_pixel_t blue  = m_pixel_rgba(0, 0, 1, 1);
    m_pixel_t black = m_pixel_rgba(0, 0, 0, 1);
    m_pixel_t white = m_pixel_rgba(1, 1, 1, 1);

    int retval = 1;
    m_image *image = m_image_new(w, h);
    if (image) {
      m_d_transform *transform = m_d_transform_rectangular(w, h, c, r);
      if (transform) {
        m_d_colour_t *colour = m_d_colour_minimal(white, black, white);
        if (colour) {
          m_d_render_scanline(image, transform, er, maxiters, colour);

          double _Complex c3, c4a, c4b, c5, c3c2, c2c3;
          m_d_nucleus(&c3, 0 + I * 1, 3, 64);
          m_d_nucleus(&c4a, 0.25 + 0.5 * I, 4, 64);
          m_d_nucleus(&c4b, 0.25 - 0.5 * I, 4, 64);
          m_d_nucleus(&c5,  0.3 + 0.3 * I, 5, 64);
          m_d_nucleus(&c3c2, c3 + I * 0.1, 6, 64);
          m_d_nucleus(&c2c3, -1 - 0.25 + 0.25 * I, 6, 64);

          double pt = 48;

          draw_internal_ray(image, transform, 1, 0, "1/2", pt, green);
          draw_internal_ray(image, transform, 1, 0, "1/3", pt, green);
          draw_internal_ray(image, transform, 1, 0, "1/4", pt, green);
          draw_internal_ray(image, transform, 1, 0, "1/5", pt, green);
          draw_internal_ray(image, transform, 1, 0, "3/4", pt, green);
          draw_internal_ray(image, transform, 2, -1, "0/1", pt, green);
          draw_internal_ray(image, transform, 2, -1, "1/3", pt, green);
          draw_internal_ray(image, transform, 3, c3, "0/1", 0.7 * pt, green);
          draw_internal_ray(image, transform, 3, c3, "1/2", 0.7 * pt, green);
          draw_internal_ray(image, transform, 3, c3, "1/3", 0.7 * pt, green);
```

```
        draw_internal_ray(image, transform, 3, c3, "1/4", 0.7 * pt, green);
        draw_internal_ray(image, transform, 3, c3, "3/4", 0.7 * pt, green);

        draw_external_ray(image, transform, ".(01)", red, 0, 0);
        draw_external_ray(image, transform, ".(10)", red, 0, 0);
        draw_external_ray(image, transform, ".(001)", red, 32, 32);
        draw_external_ray(image, transform, ".(010)", red, -48, 0);
        draw_external_ray(image, transform, ".(011)", red, 0, 0);
        draw_external_ray(image, transform, ".(100)", red, 0, 0);
        draw_external_ray(image, transform, ".(0001)", red, 0, -16);
        draw_external_ray(image, transform, ".(0010)", red, 16, 16);
        draw_external_ray(image, transform, ".(1101)", red, 0, 0);
        draw_external_ray(image, transform, ".(1110)", red, 0, 0);
        draw_external_ray(image, transform, ".(00001)", red, 0, 0);
        draw_external_ray(image, transform, ".(00010)", red, 0, 16);
        draw_external_ray(image, transform, ".(001010)", red, -32, -32);
        draw_external_ray(image, transform, ".(010001)", red, 0, 0);
        draw_external_ray(image, transform, ".(010110)", red, 0, 0);
        draw_external_ray(image, transform, ".(011001)", red, 0, 0);
        draw_external_ray(image, transform, ".(001001010)", red, -48, -48);
        draw_external_ray(image, transform, ".(001010001)", red, 0, 0);
        draw_external_ray(image, transform, ".(001001001010)", red, 0, 0);
        draw_external_ray(image, transform, ".(001001010001)", red, -16, -16);
        draw_external_ray(image, transform, ".(010010001010)", red, 32, 0);
        draw_external_ray(image, transform, ".(010010010001)", red, 0, 0);

        draw_label(image, transform,  0,   "1", 6 * pt, blue);
        draw_label(image, transform, -1,   "2", 3 * pt, blue);
        draw_label(image, transform, c3,   "3", 2 * pt, blue);
        draw_label(image, transform, c4a,  "4", 1.5 * pt, blue);
        draw_label(image, transform, c4b,  "4", 1.5 * pt, blue);
        draw_label(image, transform, c5,   "5", pt, blue);
        draw_label(image, transform, c2c3, "6", pt, blue);
        draw_label(image, transform, c3c2, "6", pt, blue);

        m_image_save_png(image, filename);
        retval = 0;
        m_d_colour_delete(colour);
      }
      m_d_transform_delete(transform);
    }
    m_image_delete(image);
  }
  return retval;
}
```

24 c/bin/m-subwake-diagram-b.c

```
#include <mandelbrot-graphics.h>
#include <mandelbrot-numerics.h>
#include <mandelbrot-symbolics.h>
#include <cairo.h>

const double twopi = 6.283185307179586;

void draw_label(m_image *image, m_d_transform *transform, double _Complex c0, ↙
    ↳ const char *text, double pt, m_pixel_t colour) {
```

```
  double _Complex c = c0;
  double _Complex dc = 1;
  m_d_transform_reverse(transform, &c, &dc);
  cairo_surface_t *surface = m_image_surface(image);
  cairo_t *cr = cairo_create(surface);
  cairo_select_font_face(cr, "LMSans10", CAIRO_FONT_SLANT_NORMAL, ↙
    ↳ CAIRO_FONT_WEIGHT_NORMAL);
  cairo_set_font_size(cr, pt);
  cairo_text_extents_t te;
  cairo_text_extents(cr, text, &te);
  cairo_move_to(cr, creal(c) - te.x_bearing - te.width / 2, cimag(c) - te.↙
    ↳ y_bearing - te.height / 2);
  cairo_text_path(cr, text);
  cairo_set_source_rgba(cr, m_pixel_red(colour), m_pixel_green(colour), ↙
    ↳ m_pixel_blue(colour), m_pixel_alpha(colour));
  cairo_fill(cr);
  cairo_destroy(cr);
}

void draw_internal_ray(m_image *image, m_d_transform *transform, int period, ↙
  ↳ double _Complex nucleus, const char *angle, double pt, m_pixel_t colour) {
  int steps = 128;
  mpq_t theta;
  mpq_init(theta);
  mpq_set_str(theta, angle, 10);
  mpq_canonicalize(theta);
  double a = twopi * mpq_get_d(theta);
  mpq_clear(theta);
  double _Complex interior = cos(a) + I * sin(a);

  double _Complex cl = 0, cl2 = 0;
  double _Complex c = nucleus;
  double _Complex z = c;
  cairo_surface_t *surface = m_image_surface(image);
  cairo_t *cr = cairo_create(surface);
  cairo_set_source_rgba(cr, m_pixel_red(colour), m_pixel_green(colour), ↙
    ↳ m_pixel_blue(colour), m_pixel_alpha(colour));
  for (int i = 0; i < steps; ++i) {
    if (2 * i == steps) {
      cl = c;
    }
    if (2 * i == steps + 2) {
      cl2 = c;
    }
    double radius = (i + 0.5) / steps;
    m_d_interior(&z, &c, z, c, radius * interior, period, 64);
    double _Complex pc = c;
    double _Complex pdc = 1;
    m_d_transform_reverse(transform, &pc, &pdc);
    if (i == 0) {
      cairo_move_to(cr, creal(pc), cimag(pc));
    } else {
      cairo_line_to(cr, creal(pc), cimag(pc));
    }
  }
  cairo_stroke(cr);
  if (a != 0) {
```

```
    double t = carg(cl2 - cl);
    cairo_save(cr);
    double _Complex dcl = 1;
    m_d_transform_reverse(transform, &cl, &dcl);
    cairo_translate(cr, creal(cl), cimag(cl));
    cairo_rotate(cr, -t);
    cairo_translate(cr, 0, -pt/3);
    cairo_select_font_face(cr, "LMSans10", CAIRO_FONT_SLANT_NORMAL, ↙
        ↳ CAIRO_FONT_WEIGHT_NORMAL);
    cairo_set_font_size(cr, pt);
    cairo_text_path(cr, angle);
    cairo_fill(cr);
    cairo_restore(cr);
  }
  cairo_destroy(cr);
}

void draw_external_ray(m_image *image, m_d_transform *transform, const char *↙
    ↳ angle, m_pixel_t colour, double dx, double dy, double _Complex c0, double ↙
    ↳ r0) {
  int maxiters = 1024;

  m_block blo, bhi;
  m_block_init(&blo);
  m_block_init(&bhi);
  m_block_from_string(&blo, "011");
  m_block_from_string(&bhi, "100");
  m_binangle btheta0;
  m_binangle_init(&btheta0);
  m_binangle_from_string(&btheta0, angle);
  m_binangle btheta;
  m_binangle_init(&btheta);
  m_binangle_tune(&btheta, &btheta0, &blo, &bhi);
  m_binangle_clear(&btheta0);
  m_block_clear(&blo);
  m_block_clear(&bhi);
  char angle2[m_binangle_strlen(&btheta) + 1];
  m_binangle_to_string(angle2, &btheta);

  mpq_t qtheta;
  mpq_init(qtheta);
  m_binangle_to_rational(qtheta, &btheta);
  m_binangle_clear(&btheta);
  m_d_exray_in *ray = m_d_exray_in_new(qtheta, 8);
  mpq_clear(qtheta);

  cairo_surface_t *surface = m_image_surface(image);
  cairo_t *cr = cairo_create(surface);
  cairo_set_source_rgba(cr, m_pixel_red(colour), m_pixel_green(colour), ↙
      ↳ m_pixel_blue(colour), m_pixel_alpha(colour));
  bool first = true;
  for (int i = 0; i < maxiters; ++i) {
    if (m_failed == m_d_exray_in_step(ray, 64)) {
      break;
    }
    double _Complex c = m_d_exray_in_get(ray);
    if (cabs(c - c0) > r0) {
```

```
      continue;
    }
    double t = carg(c - c0);
    double _Complex dc = 1;
    m_d_transform_reverse(transform, &c, &dc);
    if (first) {
      cairo_save(cr);
      cairo_translate(cr, creal(c) + dx, cimag(c) + dy);
      cairo_rotate(cr, -t);
      cairo_select_font_face(cr, "LMMono10", CAIRO_FONT_SLANT_NORMAL, ↙
          ↳ CAIRO_FONT_WEIGHT_NORMAL);
      cairo_set_font_size(cr, 48);
      cairo_text_path(cr, angle2);
      cairo_fill(cr);
      cairo_restore(cr);
      cairo_move_to(cr, creal(c) + dx, cimag(c) + dy);
      first = false;
    } else {
      cairo_line_to(cr, creal(c), cimag(c));
    }
  }
  cairo_stroke(cr);
  cairo_destroy(cr);
}

int main(int argc, char **argv) {
  (void) argc;
  (void) argv;
  int w = 4096;
  int h = 4096;

  int p = 3;
  double _Complex c1, c2, c3, c4a, c4b, c5, c3c2, c2c3;
  m_d_nucleus(&c1, -2, p * 1, 64);
  double size = cabs(m_d_size(c1, p * 1));
  m_d_nucleus(&c2, c1 - size, p * 2, 64);
  m_d_nucleus(&c3, c1 + I * size, p * 3, 64);
  m_d_nucleus(&c4a, c1 + size * 0.25 + size * 0.5 * I, p * 4, 64);
  m_d_nucleus(&c4b, c1 + size * 0.25 - size * 0.5 * I, p * 4, 64);
  m_d_nucleus(&c5,  c1 + size * 0.3 + size * 0.3 * I, p * 5, 64);
  m_d_nucleus(&c3c2, c3 + size * I * 0.1, p * 6, 64);
  m_d_nucleus(&c2c3, c2 - size * 0.25 + size * 0.25 * I, p * 6, 64);

  complex double c = (c1 + 3 * c2) / 4;
  double r = 3 * size;
  double r0 = sqrt(2) * size;
  double er = 600;
  int maxiters = 8192;
  const char *filename = "subwake-diagram-b.png";

  m_pixel_t red   = m_pixel_rgba(1, 0, 0, 1);
  m_pixel_t green = m_pixel_rgba(0, 0.5, 0, 1);
  m_pixel_t blue  = m_pixel_rgba(0, 0, 1, 1);
  m_pixel_t black = m_pixel_rgba(0, 0, 0, 1);
  m_pixel_t white = m_pixel_rgba(1, 1, 1, 1);
  double pt = 48 * 1.75 / 3;
```

```
170     int retval = 1;
        m_image *image = m_image_new(w, h);
        if (image) {
          m_d_transform *transform = m_d_transform_rectangular(w, h, c, r);
          if (transform) {
175         m_d_colour_t *colour = m_d_colour_minimal(white, black, white);
            if (colour) {
              m_d_render_scanline(image, transform, er, maxiters, colour);

              draw_internal_ray(image, transform, p * 1, c1, "1/2", pt, green);
180           draw_internal_ray(image, transform, p * 1, c1, "1/3", pt, green);
              draw_internal_ray(image, transform, p * 1, c1, "1/4", pt, green);
              draw_internal_ray(image, transform, p * 1, c1, "1/5", pt, green);
              draw_internal_ray(image, transform, p * 1, c1, "3/4", pt, green);
              draw_internal_ray(image, transform, p * 2, c2, "0/1", pt, green);
185           draw_internal_ray(image, transform, p * 2, c2, "1/3", pt, green);
              draw_internal_ray(image, transform, p * 3, c3, "0/1", 0.7 * pt, green);
              draw_internal_ray(image, transform, p * 3, c3, "1/2", 0.7 * pt, green);
              draw_internal_ray(image, transform, p * 3, c3, "1/3", 0.7 * pt, green);
              draw_internal_ray(image, transform, p * 3, c3, "1/4", 0.7 * pt, green);
190           draw_internal_ray(image, transform, p * 3, c3, "3/4", 0.7 * pt, green);

              draw_external_ray(image, transform, ".(0)", red, 0, 0, c, r0);
              draw_external_ray(image, transform, ".(1)", red, 0, 0, c, r0);
              draw_external_ray(image, transform, ".(10)", red, 0, 0, c, r0);
195           draw_external_ray(image, transform, ".(001)", red, 32, 32 - 32, c, r0);
              draw_external_ray(image, transform, ".(010)", red, -48 - 16, -32, c, r0)↙
                  ↳ ;
              draw_external_ray(image, transform, ".(011)", red, 0, 16, c, r0);
              draw_external_ray(image, transform, ".(100)", red, 0, 0, c, r0);
              draw_external_ray(image, transform, ".(0001)", red, 0, -16, c, r0);
200           draw_external_ray(image, transform, ".(0010)", red, 48, 48 - 32, c, r0);
              draw_external_ray(image, transform, ".(1101)", red, 0, 0, c, r0);
              draw_external_ray(image, transform, ".(1110)", red, 0, 0, c, r0);
              draw_external_ray(image, transform, ".(00001)", red, 0, 0, c, r0);
              draw_external_ray(image, transform, ".(00010)", red, 32, 32, c, r0);
205           draw_external_ray(image, transform, ".(001010)", red, -64, -64 - 32, c, ↙
                  ↳ r0);
              draw_external_ray(image, transform, ".(010001)", red, 48, -32, c, r0);
              draw_external_ray(image, transform, ".(010110)", red, 0, 0, c, r0);
              draw_external_ray(image, transform, ".(011001)", red, 0, -16, c, r0);
              draw_external_ray(image, transform, ".(001001010)", red, -64, -64 - 32, ↙
                  ↳ c, r0);
210           draw_external_ray(image, transform, ".(001010001)", red, -32, -32 - 32, ↙
                  ↳ c, r0);
              draw_external_ray(image, transform, ".(001001001010)", red, 0, 0 - 32, c↙
                  ↳ , r0);
              draw_external_ray(image, transform, ".(001001010001)", red, -32, -32 - ↙
                  ↳ 32, c, r0);
              draw_external_ray(image, transform, ".(010010001010)", red, 48 - 16, ↙
                  ↳ -32, c, r0);
              draw_external_ray(image, transform, ".(010010010001)", red, 0 - 16, -32,↙
                  ↳  c, r0);
215
              draw_label(image, transform, c1,   "3", 6 * pt, blue);
              draw_label(image, transform, c2,   "6", 3 * pt, blue);
              draw_label(image, transform, c3,   "9", 2 * pt, blue);
```

```
        draw_label(image, transform, c4a,  "12", 1.5 * pt, blue);
        draw_label(image, transform, c4b,  "12", 1.5 * pt, blue);
        draw_label(image, transform, c5,   "15", pt, blue);
        draw_label(image, transform, c2c3, "18", pt, blue);
        draw_label(image, transform, c3c2, "18", pt, blue);

        m_image_save_png(image, filename);
        retval = 0;
        m_d_colour_delete(colour);
      }
      m_d_transform_delete(transform);
    }
    m_image_delete(image);
  }
  return retval;
}
```

25 c/bin/m-subwake-diagram-c.c

```
#include <mandelbrot-graphics.h>
#include <mandelbrot-numerics.h>
#include <mandelbrot-symbolics.h>
#include <cairo.h>
#include <stdio.h>

const double twopi = 6.283185307179586;

void draw_label(m_image *image, m_d_transform *transform, double _Complex c0, const char *text, double pt, m_pixel_t colour, double angle) {
  double _Complex c = c0;
  double _Complex dc = 1;
  m_d_transform_reverse(transform, &c, &dc);
  cairo_surface_t *surface = m_image_surface(image);
  cairo_t *cr = cairo_create(surface);
  cairo_select_font_face(cr, "LMSans10", CAIRO_FONT_SLANT_NORMAL, CAIRO_FONT_WEIGHT_NORMAL);
  cairo_set_font_size(cr, pt);
  cairo_text_extents_t te;
  cairo_text_extents(cr, text, &te);
  cairo_translate(cr, creal(c), cimag(c));
  cairo_rotate(cr, angle);
  cairo_translate(cr, - te.x_bearing - te.width / 2, - te.y_bearing - te.height / 2);
  cairo_text_path(cr, text);
  cairo_set_source_rgba(cr, m_pixel_red(colour), m_pixel_green(colour), m_pixel_blue(colour), m_pixel_alpha(colour));
  cairo_fill(cr);
  cairo_destroy(cr);
}

void draw_external_ray(m_image *image, m_d_transform *transform, const char *angle, m_pixel_t colour, double pt, m_pixel_t lcolour, bool plabel, m_pixel_t acolour, const char *addr) {
  int maxiters = 1024;

  m_binangle btheta;
  m_binangle_init(&btheta);
```

```
  m_binangle_from_string(&btheta, angle);
  int period = btheta.per.length;

  mpq_t qtheta;
  mpq_init(qtheta);
  m_binangle_to_rational(qtheta, &btheta);
  m_binangle_clear(&btheta);
  m_d_exray_in *ray = m_d_exray_in_new(qtheta, 8);
  mpq_clear(qtheta);

  cairo_surface_t *surface = m_image_surface(image);
  cairo_t *cr = cairo_create(surface);
  cairo_set_source_rgba(cr, m_pixel_red(colour), m_pixel_green(colour), ↙
      ↳ m_pixel_blue(colour), m_pixel_alpha(colour));
  bool first = true;
  double _Complex c1 = 0;
  for (int i = 0; i < maxiters; ++i) {
    if (m_failed == m_d_exray_in_step(ray, 64)) {
      break;
    }
    double _Complex c = m_d_exray_in_get(ray);
    c1 = c;
    double _Complex dc = 1;
    double t = cimag(c) > 0 ? twopi / 4 : -twopi / 4;
    m_d_transform_reverse(transform, &c, &dc);
    double h = fabs(cimag(c) - m_image_get_height(image) / 2);
    if (h > m_image_get_height(image) / 6) {
      continue;
    }
    if (first) {
      cairo_save(cr);
      cairo_translate(cr, creal(c), cimag(c));
      cairo_rotate(cr, -t);
      cairo_select_font_face(cr, "LMMono10", CAIRO_FONT_SLANT_NORMAL, ↙
          ↳ CAIRO_FONT_WEIGHT_NORMAL);
      cairo_set_font_size(cr, 48);
      cairo_text_path(cr, angle);
      cairo_fill(cr);
      cairo_restore(cr);
      cairo_move_to(cr, creal(c), cimag(c));
      first = false;
    } else {
      cairo_line_to(cr, creal(c), cimag(c));
    }
  }
  cairo_stroke(cr);
  cairo_destroy(cr);

  if (plabel) {
    double _Complex nucleus;
    m_d_nucleus(&nucleus, c1, period, 64);
    char speriod[100];
    snprintf(speriod, 100, "%d", period);
    draw_label(image, transform, nucleus, speriod, pt, lcolour, 0);
    draw_label(image, transform, nucleus + 0.005 + I * 0.005, addr, pt / 2, ↙
        ↳ acolour, -twopi / 12);
  }
```

```
}

int main(int argc, char **argv) {
  (void) argc;
  (void) argv;
  int w = 16384;
  int h = 1024;

  double _Complex c3, c4, c5, c6, c7, c8;
  m_d_nucleus(&c3, -2, 3, 64);
  m_d_nucleus(&c4, -2, 4, 64);
  m_d_nucleus(&c5, -2, 5, 64);
  m_d_nucleus(&c6, -2, 6, 64);
  m_d_nucleus(&c7, -2, 7, 64);
  m_d_nucleus(&c8, -1.4, 8, 64);

  double er = 600;
  int maxiters = 8192;
  const char *filename = "subwake-diagram-c.png";

  m_pixel_t red   = m_pixel_rgba(1, 0, 0, 1);
  m_pixel_t green = m_pixel_rgba(0, 0.5, 0, 1);
  m_pixel_t blue  = m_pixel_rgba(0, 0, 1, 1);
  m_pixel_t black = m_pixel_rgba(0, 0, 0, 1);
  m_pixel_t white = m_pixel_rgba(1, 1, 1, 1);
  double pt = 96;

  int retval = 1;
  m_image *image = m_image_new(w, h);
  if (image) {
    double _Complex c = (c7 + c8) / 2;
    double r = (2.03 + creal(c)) * h / (double) w;
    m_d_transform *transform = m_d_transform_rectangular(w, h, c, r);

    if (transform) {
      m_d_colour_t *colour = m_d_colour_minimal(white, black, white);
      if (colour) {
        m_d_render_scanline(image, transform, er, maxiters, colour);
        /*
        $ cat feature-database.csv | grep True | grep ^[1-7], | grep -v / | (
          IFS=,
          while read p i a n1 d1 n2 d2 rest
          do
            echo ", { \"`m-binangle-from-rational $n1/$d1`\", \"`m-binangle-from
                ↪ -rational $n2/$d2`\", \"$a\" }"
          done
        )
        */
        const char *angles[19][3] =
{ { ".(011)", ".(100)", "1 2 3" }
, { ".(0111)", ".(1000)", "1 2 3 4" }
, { ".(01111)", ".(10000)", "1 2 3 4 5" }
, { ".(01110)", ".(10001)", "1 2 3 5" }
, { ".(01101)", ".(10010)", "1 2 4 5" }
, { ".(011111)", ".(100000)", "1 2 3 4 5 6" }
, { ".(011110)", ".(100001)", "1 2 3 4 6" }
, { ".(011101)", ".(100010)", "1 2 3 5 6" }
```

```
    , { ".(011010)", ".(100101)", "1 2 4 6" }
    , { ".(0111111)", ".(1000000)", "1 2 3 4 5 6 7" }
145 , { ".(0111110)", ".(1000001)", "1 2 3 4 5 7" }
    , { ".(0111101)", ".(1000010)", "1 2 3 4 6 7" }
    , { ".(0111100)", ".(1000011)", "1 2 3 4 7" }
    , { ".(0111011)", ".(1000100)", "1 2 3 5 6 7" }
    , { ".(0111010)", ".(1000101)", "1 2 3 5 7" }
150 , { ".(0111001)", ".(1000110)", "1 2 3 6 7" }
    , { ".(0110110)", ".(1001001)", "1 2 4 5 7" }
    , { ".(0110101)", ".(1001010)", "1 2 4 6 7" }
    , { ".(01101001)", ".(10010110)", "1 2 4 8" }
    };
155         for (int a = 0; a < 19; ++a) {
              for (int b = 0; b < 2; ++b) {
                draw_external_ray(image, transform, angles[a][b], red, pt, blue, b ↙
                    ↳ == 0, green, angles[a][2]);
              }
            }
160
            m_image_save_png(image, filename);
            retval = 0;
            m_d_colour_delete(colour);
          }
165       m_d_transform_delete(transform);
        }
        m_image_delete(image);
      }
      return retval;
170 }
```

26 c/bin/m-warped-midgets.c

```
#include <complex.h>
#include <math.h>
#include <stdio.h>
#include <gmp.h>
 5 #include <mandelbrot-symbolics.h>
#include <mandelbrot-numerics.h>
#include <mandelbrot-graphics.h>

int main(int argc, char **argv)
10 {
  (void) argc;
  (void) argv;
  int w = 256;
  int h = 256;
15   int sharpness = 4;
  double er = 600;
  int maxiters = 1 << 18;
  m_image *image = m_image_new(w, h);
  m_pixel_t red   = m_pixel_rgba(1, 0, 0, 1);
20   m_pixel_t black = m_pixel_rgba(0, 0, 0, 1);
  m_pixel_t white = m_pixel_rgba(1, 1, 1, 1);
  mpq_t angle;
  mpq_init(angle);
  m_binangle bangle;
25   m_binangle_init(&bangle);
```

```
  if (image)
  {
    cairo_surface_t *surface = m_image_surface(image);
    cairo_t *cr = cairo_create(surface);
    cairo_select_font_face(cr, "LMSans10", CAIRO_FONT_SLANT_NORMAL, ↙
        ↳ CAIRO_FONT_WEIGHT_BOLD);
    cairo_set_font_size(cr, 24);
    cairo_set_source_rgba(cr, 1, 0, 0, 1);
    m_d_colour_t *colour = m_d_colour_minimal(red, black, white);
    if (colour) {
      for (int period = 4; period < 1 << 14; period <<= 1)
      {
        mpz_set_ui(bangle.pre.bits, 0); bangle.pre.length = 0;
        mpz_set_ui(bangle.per.bits, 3); bangle.per.length = period;
        m_binangle_to_rational(angle, &bangle);
        complex double ray = m_d_exray_in_do(angle, sharpness, 8 * sharpness * ↙
            ↳ period, 64);
        complex double mc = 0;
        m_d_nucleus(&mc, ray, period, 64);
        double r = cabs(m_d_size(mc, period)) * 2;
        printf("%7d %.18e + %.18e i @ %.3e\n", period, creal(mc), cimag(mc), r);
        m_d_transform *transform = m_d_transform_rectangular(w, h, mc, r);
        if (transform)
        {
          m_d_render_scanline(image, transform, er, maxiters, colour);
          m_image_dirty(image);
          cairo_move_to(cr, 24, 24);
          char text[100];
          snprintf(text, 100, "%d", period);
          cairo_show_text(cr, text);
          cairo_fill(cr);
          m_image_flush(image);
          char filename[100];
          snprintf(filename, 100, "%06d.png", period);
          m_image_save_png(image, filename);
          m_d_transform_delete(transform);
        }
      }
      m_d_colour_delete(colour);
    }
    m_image_delete(image);
  }
  m_binangle_clear(&bangle);
  mpq_clear(angle);
  return 0;
}
```

27 c/include/mandelbrot-graphics.h

```
#ifndef MANDELBROT_GRAPHICS_H
#define MANDELBROT_GRAPHICS_H 1

#include <cairo.h>
#include <complex.h>
#include <math.h>
#include <stdbool.h>
#include <stdint.h>
```

```
#include <stdlib.h>
#include <gmp.h>
#include <mpfr.h>
#include <mpc.h>
#include <mandelbrot-numerics.h>

#ifdef __cplusplus
extern "C"
{
#endif

/* functions returning bool return true for success, false for failure */

enum m_compute_t {
  m_unknown,
  m_interior,
  m_exterior
};
typedef enum m_compute_t m_compute_t;

/* double precision: m_d_*()  */

struct m_d_compute;
typedef struct m_d_compute m_d_compute;

extern m_d_compute *m_d_compute_alloc(int npartials);
extern void m_d_compute_free(m_d_compute *px);
extern void m_d_compute_init(m_d_compute *px, m_compute_t bias, double er, ↙
    ↳ double _Complex c, m_period_filter_t *filter);
extern void m_d_compute_clear(m_d_compute *px);
extern bool m_d_compute_step(m_d_compute *px, int steps);
extern m_compute_t m_d_compute_get_tag(const m_d_compute *px);
extern int m_d_compute_get_n(const m_d_compute *px);
extern int m_d_compute_get_p(const m_d_compute *px);
extern int m_d_compute_get_q(const m_d_compute *px);
extern double _Complex m_d_compute_get_c(const m_d_compute *px);
extern double _Complex m_d_compute_get_z(const m_d_compute *px);
extern double _Complex m_d_compute_get_dc(const m_d_compute *px);
extern double _Complex m_d_compute_get_dz(const m_d_compute *px);
extern double _Complex m_d_compute_get_zp(const m_d_compute *px);
extern double _Complex m_d_compute_get_zq(const m_d_compute *px);
extern double m_d_compute_get_de(const m_d_compute *px);

struct m_d_transform;
typedef struct m_d_transform m_d_transform;
typedef void (m_d_transform_f)(void *userdata, double _Complex *c, double ↙
    ↳ _Complex *dc);
struct m_d_transform {
  m_d_transform_f *forward;
  m_d_transform_f *reverse;
};

extern void m_d_transform_forward(m_d_transform *t, double _Complex *c, double ↙
    ↳ _Complex *dc);
extern void m_d_transform_reverse(m_d_transform *t, double _Complex *c, double ↙
    ↳ _Complex *dc);
extern void m_d_transform_delete(m_d_transform *t);
```

```
extern m_d_transform *m_d_transform_compose(m_d_transform *s, m_d_transform *t);
extern m_d_transform *m_d_transform_invert(m_d_transform *t);
extern m_d_transform *m_d_transform_rectangular(int w, int h, double _Complex c,↙
    ↳  double r);
extern m_d_transform *m_d_transform_linear(double _Complex add, double _Complex ↙
    ↳ mul);
extern m_d_transform *m_d_transform_moebius(const m_d_mat2 *m);
extern m_d_transform *m_d_transform_moebius3(double _Complex zero, double ↙
    ↳ _Complex one, double _Complex infinity);
extern m_d_transform *m_d_transform_moebius6(double _Complex a0, double _Complex↙
    ↳  a1, double _Complex a2, double _Complex b0, double _Complex b1, double ↙
    ↳ _Complex b2);
extern m_d_transform *m_d_transform_cardioid();
extern m_d_transform *m_d_transform_exponential(double _Complex center);

typedef uint32_t m_pixel_t;

extern double m_pixel_red(m_pixel_t p);
extern double m_pixel_green(m_pixel_t p);
extern double m_pixel_blue(m_pixel_t p);
extern double m_pixel_alpha(m_pixel_t p);
extern m_pixel_t m_pixel_rgba(double r, double g, double b, double a);
extern m_pixel_t m_pixel_hsva(double h, double s, double v, double a);
extern m_pixel_t m_pixel_mix(m_pixel_t a, m_pixel_t b, double x);
extern m_pixel_t m_pixel_blacken(m_pixel_t a, double x);
extern m_pixel_t m_pixel_whiten(m_pixel_t a, double x);

struct m_image;
typedef struct m_image m_image;

extern m_image *m_image_new(int width, int height);
extern void m_image_delete(m_image *img);
extern m_image *m_image_load_ppm(const char *filename);
extern void m_image_flush(m_image *img);
extern void m_image_dirty(m_image *img);
extern bool m_image_in_bounds(m_image *img, int x, int y);
extern void m_image_plot(m_image *img, int x, int y, m_pixel_t c);
extern m_pixel_t m_image_peek(m_image *img, int x, int y);
extern bool m_image_save_png(m_image *img, const char *filename);
extern int m_image_get_width(const m_image *img);
extern int m_image_get_height(const m_image *img);
extern cairo_surface_t *m_image_surface(m_image *img);

struct m_mipmap;
typedef struct m_mipmap m_mipmap;

extern m_mipmap *m_mipmap_new(m_image *img);
extern m_pixel_t m_mipmap_linear(const m_mipmap *m, double level, double x, ↙
    ↳ double y);
extern m_pixel_t m_mipmap_linear_linear(const m_mipmap *m, double level, double ↙
    ↳ x, double y);
extern int m_mipmap_get_levels(const m_mipmap *m);

struct m_d_colour_t;
typedef struct m_d_colour_t m_d_colour_t;
typedef m_pixel_t (m_d_colour_f)(m_d_colour_t *colour, m_d_compute *px, double ↙
    ↳ er, double _Complex dc);
```

```
struct m_d_colour_t {
  m_d_colour_f *colour;
};

extern void m_d_colour_delete(m_d_colour_t *colour);
extern m_pixel_t m_d_colour(m_d_colour_t *colour, m_d_compute *px, double er, double _Complex dc);
extern m_d_colour_t *m_d_colour_minimal(m_pixel_t interior, m_pixel_t boundary, m_pixel_t exterior);
extern double _Complex m_d_colour_exterior_coordinates(m_d_compute *px, double er);
extern double m_d_colour_grid(m_d_compute *px, double er);
extern m_d_colour_t *m_d_colour_minimal_grid(m_pixel_t interior, m_pixel_t boundary, m_pixel_t exterior, m_pixel_t grid);
extern m_d_colour_t *m_d_colour_minimal_texture(m_pixel_t interior, m_pixel_t boundary, m_mipmap *exterior);
extern m_d_colour_t *m_d_colour_domain(double hue, double sat, double val, m_period_filter_t *filter);

extern void m_d_render_scanline_filtered(m_image *image, m_d_transform *transform, double er, int maxiters, m_d_colour_t *colour, m_period_filter_t *filter);
extern void m_d_render_scanline(m_image *image, m_d_transform *transform, double er, int maxiters, m_d_colour_t *colour);

#ifdef __cplusplus
}
#endif

#endif
```

28 c/lib/Makefile

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

PKGCONFIG := PKG_CONFIG_PATH="$(prefix)/lib/pkgconfig" pkg-config
COMPILE := $(CC) -std=c99 -Wall -Wextra -pedantic -fPIC -O3 -fopenmp -pipe -g -MMD -I../include `$(PKGCONFIG) --cflags mandelbrot-numerics cairo` -c
LINK    := $(CC) -fopenmp -shared -g `$(PKGCONFIG) --libs mandelbrot-numerics cairo`
LIBRARY := libmandelbrot-graphics
OBJECTS := $(patsubst %.c,%.o,$(wildcard *.c))
DEPENDS := $(patsubst %.o,%.d,$(OBJECTS))

all: $(LIBRARY).a $(LIBRARY).so pkgconfig/mandelbrot-graphics.pc

clean:
	@echo "CLEAN" ; rm -f $(OBJECTS) $(DEPENDS) $(LIBRARY).a $(LIBRARY).so pkgconfig/mandelbrot-graphics.pc

install: $(LIBRARY).a $(LIBRARY).so ../include/mandelbrot-graphics.h pkgconfig/mandelbrot-graphics.pc
	install -d "$(prefix)/include" "$(prefix)/lib" "$(prefix)/lib/pkgconfig"
	install -m 644 -t "$(prefix)/include" ../include/mandelbrot-graphics.h
	install -m 644 -t "$(prefix)/lib" $(LIBRARY).a $(LIBRARY).so
	install -m 644 -t "$(prefix)/lib/pkgconfig" pkgconfig/mandelbrot-
```

```
            ↳ graphics.pc

$(LIBRARY).a: $(OBJECTS)
        @echo "A       $@" ; ar -rs $@ $^ || ( echo "ERROR   ar -rs $@ $^" && ↙
            ↳ false )

$(LIBRARY).so: $(OBJECTS)
        @echo "SO      $@" ; $(LINK) -o $@ $^ -lmpc -lmpfr -lgmp -lm || ( echo "↙
            ↳ ERROR   $(LINK) -o $@ $^ -lmpc -lmpfr -lgmp -lm" && false )

%.o: %.c
        @echo "O       $@" ; $(COMPILE) -o $@ $< || ( echo "ERROR   $(COMPILE) -↙
            ↳ o $@ $<" && false )

pkgconfig/mandelbrot-graphics.pc: pkgconfig/mandelbrot-graphics.pc.in
        @echo "PC      $@" ; ( echo "prefix=$(prefix)" ; cat pkgconfig/↙
            ↳ mandelbrot-graphics.pc.in ) > pkgconfig/mandelbrot-graphics.pc || ↙
            ↳ ( echo 'ERROR    ( echo "prefix=$(prefix)" ; cat pkgconfig/↙
            ↳ mandelbrot-graphics.pc.in ) > pkgconfig/mandelbrot-graphics.pc' &&↙
            ↳  false )

.SUFFIXES:
.PHONY: all clean install

-include $(DEPENDS)
```

29 c/lib/m_d_colour.c

```
#include <mandelbrot-graphics.h>
#include "m_d_util.h"

extern void m_d_colour_delete(m_d_colour_t *colour) {
  free(colour);
}

extern m_pixel_t m_d_colour(m_d_colour_t *colour, m_d_compute *px, double er, ↙
    ↳ double _Complex dc) {
  return colour->colour(colour, px, er, dc);
}


struct m_d_colour_minimal_t {
  m_d_colour_t colour;
  m_pixel_t interior;
  m_pixel_t boundary;
  m_pixel_t exterior;
};
typedef struct m_d_colour_minimal_t m_d_colour_minimal_t;

static m_pixel_t m_d_colour_minimal_do(m_d_colour_t *colour, m_d_compute *px, ↙
    ↳ double er, double _Complex dc) {
  (void) er;
  m_d_colour_minimal_t *c = (m_d_colour_minimal_t *) colour;
  m_pixel_t base = c->boundary;
  switch (m_d_compute_get_tag(px)) {
    case m_exterior: base = c->exterior; break;
    case m_interior: base = c->interior; break;
```

```
    default:          base = c->boundary; break;
  }
  double de = clamp(m_d_compute_get_de(px) / cabs(dc), 0, 8);
  return m_pixel_mix(c->boundary, base, tanh(de));
}

extern m_d_colour_t *m_d_colour_minimal(m_pixel_t interior, m_pixel_t boundary, ↙
    ↳ m_pixel_t exterior) {
  m_d_colour_minimal_t *c = malloc(sizeof(*c));
  c->colour.colour = m_d_colour_minimal_do;
  c->interior = interior;
  c->boundary = boundary;
  c->exterior = exterior;
  return &c->colour;
}


extern double _Complex m_d_colour_exterior_coordinates(m_d_compute *px, double ↙
    ↳ er) {
  double _Complex z = m_d_compute_get_z(px);
  double r = log(cabs(z)) / log(er) - 1.0;
  double a = fmod(carg(z) / twopi + 1.0, 1.0);
  return a + I * r;
}


extern double m_d_colour_grid(m_d_compute *px, double er) {
  double _Complex z = m_d_colour_exterior_coordinates(px, er);
  double r = cimag(z);
  double a = creal(z);
  double k = pow(0.5, 0.5 - r);
  double w = 0.05;
  return 1.0 - (w < r && r < 1.0 - w && w * k < a && a < 1.0 - w * k);
}


struct m_d_colour_minimal_grid_t {
  m_d_colour_t colour;
  m_pixel_t interior;
  m_pixel_t boundary;
  m_pixel_t exterior;
  m_pixel_t grid;
};
typedef struct m_d_colour_minimal_grid_t m_d_colour_minimal_grid_t;

static m_pixel_t m_d_colour_minimal_grid_do(m_d_colour_t *colour, m_d_compute *↙
    ↳ px, double er, double _Complex dc) {
  (void) er;
  m_d_colour_minimal_grid_t *c = (m_d_colour_minimal_grid_t *) colour;
  m_pixel_t base = c->boundary;
  double grid = 0.0;
  switch (m_d_compute_get_tag(px)) {
    case m_exterior: base = c->exterior; grid = m_d_colour_grid(px, er); break;
    case m_interior: base = c->interior; break;
    default:         base = c->boundary; break;
  }
  base = m_pixel_mix(base, c->grid, grid);
```

```
  double de = clamp(m_d_compute_get_de(px) / cabs(dc), 0, 8);
  return m_pixel_mix(c->boundary, base, tanh(de));
}

extern m_d_colour_t *m_d_colour_minimal_grid(m_pixel_t interior, m_pixel_t ↙
    ↳ boundary, m_pixel_t exterior, m_pixel_t grid) {
  m_d_colour_minimal_grid_t *c = malloc(sizeof(*c));
  c->colour.colour = m_d_colour_minimal_grid_do;
  c->interior = interior;
  c->boundary = boundary;
  c->exterior = exterior;
  c->grid = grid;
  return &c->colour;
}

struct m_d_colour_minimal_texture_t {
  m_d_colour_t colour;
  m_pixel_t interior;
  m_pixel_t boundary;
  m_mipmap *exterior;
};
typedef struct m_d_colour_minimal_texture_t m_d_colour_minimal_texture_t;

static m_pixel_t m_d_colour_minimal_texture_do(m_d_colour_t *colour, m_d_compute↙
    ↳ *px, double er, double _Complex dc) {
  m_d_colour_minimal_texture_t *c = (m_d_colour_minimal_texture_t *) colour;
  double de = m_d_compute_get_de(px) / cabs(dc);
  m_pixel_t base = c->boundary;
  switch (m_d_compute_get_tag(px)) {
    case m_exterior: {
      double level = m_mipmap_get_levels(c->exterior) - log2(de);
      double _Complex z = m_d_colour_exterior_coordinates(px, er);
      base = m_mipmap_linear_linear(c->exterior, level, creal(z), 1 - cimag(z));
      break;
    }
    case m_interior: base = c->interior; break;
    default:         base = c->boundary; break;
  }
  return m_pixel_mix(c->boundary, base, tanh(clamp(de, 0, 8)));
}

extern m_d_colour_t *m_d_colour_minimal_texture(m_pixel_t interior, m_pixel_t ↙
    ↳ boundary, m_mipmap *exterior) {
  m_d_colour_minimal_texture_t *c = malloc(sizeof(*c));
  c->colour.colour = m_d_colour_minimal_texture_do;
  c->interior = interior;
  c->boundary = boundary;
  c->exterior = exterior;
  return &c->colour;
}

struct m_d_colour_domain_t {
  m_d_colour_t colour;
  double hue;
```

```
  double sat;
  double val;
  m_period_filter_t *filter;
};
typedef struct m_d_colour_domain_t m_d_colour_domain_t;

static m_pixel_t m_d_colour_domain_do(m_d_colour_t *c0, m_d_compute *px, double er, double _Complex dc) {
  (void) er;
  m_d_colour_domain_t *c = (m_d_colour_domain_t *) c0;
  int p = m_d_compute_get_p(px);
  double _Complex zp = m_d_compute_get_zp(px);
  if (m_d_compute_get_tag(px) == m_exterior)
  {
    if (c->filter && c->filter->reject && c->filter->reject(c->filter, p))
    {
      p = m_d_compute_get_q(px);
      zp = m_d_compute_get_zq(px);
    }
  }
  int q1 = creal(zp) > 0;
  int q2 = (cimag(zp) > 0) != q1;
  int w = 0;
  int b = 0;
  if (m_d_compute_get_tag(px) == m_exterior)
  {
    double _Complex z = m_d_compute_get_z(px);
    w = cimag(z) > 0;
    int n = m_d_compute_get_n(px);
    b = n & 1;
  }
  return m_pixel_blacken
    ( m_pixel_blacken(m_pixel_whiten(m_pixel_hsva
        ( c->hue * (p - 1)
        , c->sat * (0.8 + 0.2 * q1)
        , c->val * (0.8 + 0.2 * q2)
        , 1
        ), 0.2 * w), 0.025 * b)
    , 1 - tanh(m_d_compute_get_de(px) / cabs(dc))
    );
}

extern m_d_colour_t *m_d_colour_domain(double hue, double sat, double val, m_period_filter_t *filter)
{
  m_d_colour_domain_t *c = malloc(sizeof(*c));
  c->colour.colour = m_d_colour_domain_do;
  c->hue = hue;
  c->sat = sat;
  c->val = val;
  c->filter = filter;
  return &c->colour;
}
```

30 c/lib/m_d_compute.c

```
#include <mandelbrot-graphics.h>
```

```c
#include <mandelbrot-numerics.h>
#include "m_d_util.h"

struct m_d_partial {
  double _Complex z;
  int p;
};
typedef struct m_d_partial m_d_partial;

struct m_d_compute {
  m_compute_t tag, bias;
  m_period_filter_t *filter;
  m_d_partial *partials;
  int npartials, np, n, p, q;
  double er2, mz2, mzq2, de;
  double _Complex c, z, dc, dz, zp, zq;
};

extern m_d_compute *m_d_compute_alloc(int npartials) {
  m_d_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 = m_unknown;
  return px;
}

extern void m_d_compute_free(m_d_compute *px) {
  if (px) {
    if (px->partials) {
      free(px->partials);
    }
    free(px);
  }
}

extern void m_d_compute_init(m_d_compute *px, m_compute_t bias, double er, ↙
    ↳ double _Complex c, m_period_filter_t *filter) {
  if (! px) { return; }
  px->tag = m_unknown;
  px->bias = bias;
  px->er2 = er * er;
  px->filter = filter;
  px->mz2 = 1.0 / 0.0;
  px->mzq2 = 1.0 / 0.0;
  px->c = c;
```

```
  px->z = 0;
  px->dc = 0;
  px->dz = 0;
  px->zp = 0;
  px->zq = 0;
  px->n = 0;
  px->p = 0;
  px->q = 0;
  px->np = 0;
  px->de = -1;
}

extern void m_d_compute_clear(m_d_compute *px) {
  px->tag = m_unknown;
  px->np = 0;
}

extern bool m_d_compute_step(m_d_compute *px, int steps) {
  if (! px) {
    return false;
  }
  if (px->tag != m_unknown) {
    return true;
  }
  double er2 = px->er2;
  double _Complex c = px->c;
  double _Complex z = px->z;
  double _Complex dc = px->dc;
  double _Complex zp = px->zp;
  double _Complex zq = px->zq;
  double mz2 = px->mz2;
  double mzq2 = px->mzq2;
  int p = px->p;
  int q = px->q;
  for (int i = 1; i <= steps; ++i) {
    dc = 2 * z * dc + 1;
    z = z * z + c;
    double z2 = cabs2(z);
    if (z2 < mzq2 && px->filter && px->filter->accept && px->filter->accept(px->↙
        ↳ filter, px->n + i))
    {
      mzq2 = z2;
      q = px->n + i;
      zq = z;
    }
    if (z2 < mz2) {
      mz2 = z2;
      p = px->n + i;
      zp = z;
      if (px->bias == m_interior) {
        double _Complex dz = 0;
        double de = -1;
        if (m_d_interior_de(&de, &dz, z, c, p, 64)) {
          px->tag = m_interior;
          px->p = p;
          px->z = z;
          px->dz = dz;
```

```
            px->zp = zp;
            px->de = de;
            return true;
          }
        } else {
          if (px->partials && px->np < px->npartials) {
            px->partials[px->np].z = z;
            px->partials[px->np].p = p;
            px->np = px->np + 1;
          }
        }
      }
      if (! (z2 < er2)) {
        px->tag = m_exterior;
        px->n = px->n + i;
        px->p = p;
        px->q = q;
        px->z = z;
        px->zp = zp;
        px->zq = zq;
        px->dc = dc;
        px->de = 2 * cabs(z) * log(cabs(z)) / cabs(dc);
        return true;
      }
    }
    if (px->bias != m_interior && px->partials) {
      for (int i = 0; i < px->np; ++i) {
        z = px->partials[i].z;
        zp = z;
        int p = px->partials[i].p;
        double _Complex dz = 0;
        double de = -1;
        if (m_d_interior_de(&de, &dz, z, c, p, 64)) {
          px->tag = m_interior;
          px->p = p;
          px->z = z;
          px->dz = dz;
          px->zp = zp;
          px->de = de;
          return true;
        }
      }
    }
    px->tag = m_unknown;
    px->n = px->n + steps;
    px->p = p;
    px->q = q;
    px->mz2 = mz2;
    px->mzq2 = mzq2;
    px->z = z;
    px->dc = dc;
    px->zp = zp;
    px->zq = zq;
    return false;
  }

  extern m_compute_t m_d_compute_get_tag(const m_d_compute *px) {
```

```
  if (! px) { return m_unknown; }
  return px->tag;
}

extern int m_d_compute_get_n(const m_d_compute *px) {
  if (! px) { return 0; }
  return px->n;
}

extern int m_d_compute_get_p(const m_d_compute *px) {
  if (! px) { return 0; }
  return px->p;
}

extern int m_d_compute_get_q(const m_d_compute *px) {
  if (! px) { return 0; }
  return px->q;
}

extern double _Complex m_d_compute_get_c(const m_d_compute *px) {
  if (! px) { return 0; }
  return px->c;
}

extern double _Complex m_d_compute_get_z(const m_d_compute *px) {
  if (! px) { return 0; }
  return px->z;
}

extern double _Complex m_d_compute_get_dc(const m_d_compute *px) {
  if (! px) { return 0; }
  return px->dc;
}

extern double _Complex m_d_compute_get_dz(const m_d_compute *px) {
  if (! px) { return 0; }
  return px->dz;
}

extern double _Complex m_d_compute_get_zp(const m_d_compute *px) {
  if (! px) { return 0; }
  return px->zp;
}

extern double _Complex m_d_compute_get_zq(const m_d_compute *px) {
  if (! px) { return 0; }
  return px->zq;
}

extern double m_d_compute_get_de(const m_d_compute *px) {
  if (! px) { return 0; }
  return px->de;
}
```

31 c/lib/m_d_render_scanline.c

```
#include <mandelbrot-graphics.h>
```

```
#include "m_d_util.h"

extern void m_d_render_scanline_filtered(m_image *image, m_d_transform *↙
    ↳ transform, double er, int maxiters, m_d_colour_t *colour, ↙
    ↳ m_period_filter_t *filter)
{
  m_image_flush(image);
  int width = m_image_get_width(image);
  int height = m_image_get_height(image);
  #pragma omp parallel for schedule(dynamic, 1)
  for (int j = 0; j < height; ++j)
  {
    m_d_compute *px = m_d_compute_alloc(maxiters);
    for (int i = 0; i < width; ++i)
    {
      double _Complex c = i + I * j;
      double _Complex dc = 1;
      m_d_transform_forward(transform, &c, &dc);
      m_d_compute_init(px, m_d_compute_get_tag(px), er, c, filter);
      m_d_compute_step(px, maxiters);
      m_image_plot(image, i, j, m_d_colour(colour, px, er, dc));
    }
    m_d_compute_free(px);
  }
  m_image_dirty(image);
}

extern void m_d_render_scanline(m_image *image, m_d_transform *transform, double↙
    ↳  er, int maxiters, m_d_colour_t *colour)
{
  m_d_render_scanline_filtered(image, transform, er, maxiters, colour, 0);
}
```

32 c/lib/m_d_transform.c

```
#include <mandelbrot-graphics.h>
#include "m_d_util.h"

extern void m_d_transform_forward(m_d_transform *t, double _Complex *c, double ↙
    ↳ _Complex *dc) {
  t->forward(t, c, dc);
}

extern void m_d_transform_reverse(m_d_transform *t, double _Complex *c, double ↙
    ↳ _Complex *dc) {
  t->reverse(t, c, dc);
}

extern void m_d_transform_delete(m_d_transform *t) {
  free(t);
}

struct m_d_transform_compose_t {
  m_d_transform transform;
  m_d_transform *s;
  m_d_transform *t;
};
```

```
typedef struct m_d_transform_compose_t m_d_transform_compose_t;

static void m_d_transform_compose_forward(void *userdata, double _Complex *c, ↙
    ↳ double _Complex *dc) {
  m_d_transform_compose_t *t = userdata;
  m_d_transform_forward(t->s, c, dc);
  m_d_transform_forward(t->t, c, dc);
}

static void m_d_transform_compose_reverse(void *userdata, double _Complex *c, ↙
    ↳ double _Complex *dc) {
  m_d_transform_compose_t *t = userdata;
  m_d_transform_reverse(t->t, c, dc);
  m_d_transform_reverse(t->s, c, dc);
}

extern m_d_transform *m_d_transform_compose(m_d_transform *s, m_d_transform *t) ↙
    ↳ {
  m_d_transform_compose_t *r = malloc(sizeof(*r));
  r->transform.forward = m_d_transform_compose_forward;
  r->transform.reverse = m_d_transform_compose_reverse;
  r->s = s;
  r->t = t;
  return &r->transform;
}

extern m_d_transform *m_d_transform_invert(m_d_transform *t) {
  m_d_transform_f *f = t->forward;
  t->forward = t->reverse;
  t->reverse = f;
  return t;
}

struct m_d_transform_rectangular_t {
  m_d_transform transform;
  int width;
  int height;
  double _Complex c;
  double r;
};
typedef struct m_d_transform_rectangular_t m_d_transform_rectangular_t;

static void m_d_transform_rectangular_forward(void *userdata, double _Complex *c↙
    ↳ , double _Complex *dc) {
  m_d_transform_rectangular_t *rect = userdata;
  double _Complex c0 = *c;
  double _Complex dc0 = *dc;
  *c = rect->c + rect->r * conj(c0 - ((rect->width/2 - 0.5) + I * (rect->height↙
      ↳ /2 - 0.5))) / (rect->height/2);
  *dc = conj(dc0 * rect->r / (rect->height/2));
}

static void m_d_transform_rectangular_reverse(void *userdata, double _Complex *c↙
    ↳ , double _Complex *dc) {
  m_d_transform_rectangular_t *rect = (m_d_transform_rectangular_t *) userdata;
  double _Complex c0 = *c;
  double _Complex dc0 = *dc;
```

```
  *c = conj((c0 - rect->c) / rect->r * (rect->height/2)) + ((rect->width/2 - ↙
      ↳ 0.5) + I * (rect->height/2 - 0.5));
  *dc = conj(dc0 / rect->r * (rect->height/2));
}

extern m_d_transform *m_d_transform_rectangular(int w, int h, double _Complex c,↙
    ↳  double r) {
  m_d_transform_rectangular_t *rect = malloc(sizeof(*rect));
  rect->transform.forward = m_d_transform_rectangular_forward;
  rect->transform.reverse = m_d_transform_rectangular_reverse;
  rect->width = w;
  rect->height = h;
  rect->c = c;
  rect->r = r;
  return &rect->transform;
}


struct m_d_transform_linear_t {
  m_d_transform transform;
  double _Complex add;
  double _Complex mul;
};
typedef struct m_d_transform_linear_t m_d_transform_linear_t;

static void m_d_transform_linear_forward(void *userdata, double _Complex *c, ↙
    ↳ double _Complex *dc) {
  m_d_transform_linear_t *t = userdata;
  double _Complex c0 = *c;
  double _Complex dc0 = *dc;
  *c = t->mul * c0 + t->add;
  *dc = t->mul * dc0;
}

static void m_d_transform_linear_reverse(void *userdata, double _Complex *c, ↙
    ↳ double _Complex *dc) {
  m_d_transform_linear_t *t = userdata;
  double _Complex c0 = *c;
  double _Complex dc0 = *dc;
  *c = (c0 - t->add) / t->mul;
  *dc = dc0 / t->mul;
}

extern m_d_transform *m_d_transform_linear(double _Complex add, double _Complex ↙
    ↳ mul) {
  m_d_transform_linear_t *t = malloc(sizeof(*t));
  t->transform.forward = m_d_transform_linear_forward;
  t->transform.reverse = m_d_transform_linear_reverse;
  t->add = add;
  t->mul = mul;
  return &t->transform;
}

struct m_d_transform_moebius_t {
  m_d_transform transform;
  m_d_mat2 m;
};
```

```
typedef struct m_d_transform_moebius_t m_d_transform_moebius_t;

static void m_d_transform_moebius_forward(void *userdata, double _Complex *c, ↙
    ↳ double _Complex *dc) {
  m_d_transform_moebius_t *t = userdata;
  double _Complex c0 = *c;
  double _Complex dc0 = *dc;
  double _Complex d = t->m.c * c0 + t->m.d;
  *c = (t->m.a * c0 + t->m.b) / d;
  *dc = dc0 * m_d_mat2_det(&t->m) / (d * d);
}

static void m_d_transform_moebius_reverse(void *userdata, double _Complex *c, ↙
    ↳ double _Complex *dc) {
  m_d_transform_moebius_t *t = userdata;
  double _Complex c0 = *c;
  double _Complex dc0 = *dc;
  double _Complex d = -t->m.c * c0 + t->m.a;
  *c = (t->m.d * c0 - t->m.b) / d;
  *dc = dc0 * m_d_mat2_det(&t->m) / (d * d);
}

extern m_d_transform *m_d_transform_moebius(const m_d_mat2 *m) {
  m_d_transform_moebius_t *t = malloc(sizeof(*t));
  t->transform.forward = m_d_transform_moebius_forward;
  t->transform.reverse = m_d_transform_moebius_reverse;
  m_d_mat2_set(&t->m, m);
  return &t->transform;
}

extern m_d_transform *m_d_transform_moebius3(double _Complex zero, double ↙
    ↳ _Complex one, double _Complex infinity) {
  m_d_mat2 m;
  m_d_mat2_moebius3(&m, zero, one, infinity);
  return m_d_transform_moebius(&m);
}

extern m_d_transform *m_d_transform_moebius6(double _Complex a0, double _Complex↙
    ↳  a1, double _Complex a2, double _Complex b0, double _Complex b1, double ↙
    ↳ _Complex b2) {
  m_d_mat2 ma, mb, m;
  m_d_mat2_moebius3(&ma, a0, a1, a2);
  m_d_mat2_moebius3(&mb, b0, b1, b2);
  m_d_mat2_inv(&mb, &mb);
  m_d_mat2_mul(&m, &ma, &mb);
  return m_d_transform_moebius(&m);
}

struct m_d_transform_cardioid_t {
  m_d_transform transform;
};
typedef struct m_d_transform_cardioid_t m_d_transform_cardioid_t;

static void m_d_transform_cardioid_reverse(void *userdata, double _Complex *c, ↙
    ↳ double _Complex *dc) {
  (void) userdata;
  double _Complex c0 = *c;
```

```
175   double _Complex dc0 = *dc;
      double _Complex r = csqrt(1 - 4 * c0);
      *c = r - 1;
      *dc = dc0 * -2 / r;
    }
180
    static void m_d_transform_cardioid_forward(void *userdata, double _Complex *c, ↙
        ↳ double _Complex *dc) {
      (void) userdata;
      double _Complex c0 = *c;
      double _Complex dc0 = *dc;
185   double _Complex r = c0 + 1;
      *c = (1 - r * r) / 4;
      *dc = dc0 * -r / 2;
    }

190 extern m_d_transform *m_d_transform_cardioid() {
      m_d_transform_cardioid_t *t = malloc(sizeof(*t));
      t->transform.forward = m_d_transform_cardioid_forward;
      t->transform.reverse = m_d_transform_cardioid_reverse;
      return &t->transform;
195 }

    struct m_d_transform_exponential_t {
      m_d_transform transform;
      double _Complex center;
200 };
    typedef struct m_d_transform_exponential_t m_d_transform_exponential_t;

    static void m_d_transform_exponential_reverse(void *userdata, double _Complex *c↙
        ↳ , double _Complex *dc) {
      m_d_transform_exponential_t *t = userdata;
205   double _Complex c0 = *c;
      double _Complex dc0 = *dc;
      *c = clog(c0 - t->center);
      *dc = dc0 / (c0 - t->center);
    }
210
    static void m_d_transform_exponential_forward(void *userdata, double _Complex *c↙
        ↳ , double _Complex *dc) {
      m_d_transform_exponential_t *t = userdata;
      double _Complex c0 = *c;
      double _Complex dc0 = *dc;
215   *c = cexp(c0) + t->center;
      *dc = dc0 * cexp(c0);
    }

    extern m_d_transform *m_d_transform_exponential(double _Complex center) {
220   m_d_transform_exponential_t *t = malloc(sizeof(*t));
      t->transform.forward = m_d_transform_exponential_forward;
      t->transform.reverse = m_d_transform_exponential_reverse;
      t->center = center;
      return &t->transform;
225 }
```

33 c/lib/m_d_util.h

```
#ifndef M_D_UTIL_H
#define M_D_UTIL_H 1

static inline int sgn(double z) {
  if (z > 0) { return  1; }
  if (z < 0) { return -1; }
  return 0;
}

static inline bool odd(int a) {
  return a & 1;
}

static inline double cabs2(complex double z) {
  return creal(z) * creal(z) + cimag(z) * cimag(z);
}

static inline bool cisfinite(complex double z) {
  return isfinite(creal(z)) && isfinite(cimag(z));
}

static const double pi = 3.141592653589793;
static const double twopi = 6.283185307179586;

// last . takeWhile (\x -> 2 /= 2 + x) . iterate (/2) $ 1 :: Double
static const double epsilon = 4.440892098500626e-16;

// epsilon^2
static const double epsilon2 = 1.9721522630525295e-31;

static const double phi = 1.618033988749895;

static inline double mix(double a, double b, double x) {
  return (1 - x) * a + x * b;
}

static inline double clamp(double x, double lo, double hi) {
  return fmin(fmax(x, lo), hi);
}

#endif
```

34 c/lib/m_image.c

```
#include <mandelbrot-graphics.h>
#include <stdio.h>
#include "m_d_util.h"

struct m_image {
  int width;
  int height;
  int stride;
  m_pixel_t *data;
  cairo_surface_t *surface;
};

extern m_image *m_image_new(int width, int height) {
```

```
  m_image *img = (m_image *) calloc(1, sizeof(*img));
  img->width = width;
  img->height = height;
  img->stride = cairo_format_stride_for_width(CAIRO_FORMAT_ARGB32, width);
  img->data = (m_pixel_t *) calloc(1, img->stride * height);
  img->surface = cairo_image_surface_create_for_data((unsigned char *) img->data
    , CAIRO_FORMAT_ARGB32, width, height, img->stride);
  return img;
}

extern void m_image_delete(m_image *img) {
  cairo_surface_destroy(img->surface);
  free(img->data);
  free(img);
}

extern m_image *m_image_load_ppm(const char *filename) {
  FILE *in = fopen(filename, "rb");
  int w = 0, h = 0;
  fscanf(in, "P6\n%d %d 255", &w, &h);
  if (! (w > 0 && h > 0 && '\n' == fgetc(in))) {
    fclose(in);
    return 0;
  }
  unsigned char *ppm = malloc(3 * w * h);
  if (! (ppm && 1 == fread(ppm, 3 * w * h, 1, in))) {
    fclose(in);
    return 0;
  }
  fclose(in);
  m_image *img = m_image_new(w, h);
  if (! img) {
    free(ppm);
    return 0;
  }
  #pragma omp parallel for
  for (int j = 0; j < h; ++j) {
    for (int i = 0; i < w; ++i) {
      m_image_plot(img, i, j, m_pixel_rgba
        ( ppm[3 * (w * j + i) + 0] / 255.0
        , ppm[3 * (w * j + i) + 1] / 255.0
        , ppm[3 * (w * j + i) + 2] / 255.0
        , 1.0
        ));
    }
  }
  m_image_dirty(img);
  free(ppm);
  return img;
}

extern cairo_surface_t *m_image_surface(m_image *img) {
  return img->surface;
}

extern void m_image_flush(m_image *img) {
  cairo_surface_flush(img->surface);
```

```
}

extern void m_image_dirty(m_image *img) {
  cairo_surface_mark_dirty(img->surface);
}

extern bool m_image_in_bounds(m_image *img, int x, int y) {
  return 0 <= x && x < img->width && 0 <= y && y < img->height;
}

extern void m_image_plot(m_image *img, int x, int y, m_pixel_t c) {
  img->data[y * (img->stride >> 2) + x] = c;
}

extern m_pixel_t m_image_peek(m_image *img, int x, int y) {
  return img->data[y * (img->stride >> 2) + x];
}

extern bool m_image_save_png(m_image *img, const char *filename) {
  return CAIRO_STATUS_SUCCESS == cairo_surface_write_to_png(img->surface, ↙
      ↳ filename);
}

extern int m_image_get_width(const m_image *img) {
  return img->width;
}

extern int m_image_get_height(const m_image *img) {
  return img->height;
}
```

35 c/lib/m_mipmap.c

```
#include <mandelbrot-graphics.h>
#include "m_d_util.h"

struct m_mipmap {
  int levels;
  m_image **images;
};

extern m_mipmap *m_mipmap_new(m_image *img) {
  int levels = 1 + round(log2(m_image_get_width(img)));
  int size = 1 << (levels - 1);
  if (! (m_image_get_width(img) == size && m_image_get_height(img) == size)) {
    return 0;
  }
  m_mipmap *m = malloc(sizeof(*m));
  m->levels = levels;
  m->images = malloc(m->levels * sizeof(*m->images));
  m->images[0] = img;
  for (int l = 1; l < m->levels; ++l) {
    int w = m_image_get_width(m->images[l-1])/2;
    int h = m_image_get_height(m->images[l-1])/2;
    m->images[l] = m_image_new(w, h);
    #pragma omp parallel for
    for (int j = 0; j < h; ++j) {
```

```
25        for (int i = 0; i < w; ++i) {
            m_pixel_t p00 = m_image_peek(m->images[l-1], 2 * i + 0, 2 * j + 0);
            m_pixel_t p10 = m_image_peek(m->images[l-1], 2 * i + 1, 2 * j + 0);
            m_pixel_t p01 = m_image_peek(m->images[l-1], 2 * i + 0, 2 * j + 1);
            m_pixel_t p11 = m_image_peek(m->images[l-1], 2 * i + 1, 2 * j + 1);
30          m_pixel_t p0 = m_pixel_mix(p00, p01, 0.5);
            m_pixel_t p1 = m_pixel_mix(p10, p11, 0.5);
            m_pixel_t p = m_pixel_mix(p0, p1, 0.5);
            m_image_plot(m->images[l], i, j, p);
          }
35      }
        m_image_dirty(m->images[l]);
      }
      return m;
    }
40
    extern m_pixel_t m_mipmap_linear(const m_mipmap *m, double level, double x, ↙
        ↳ double y) {
      int l = clamp(round(level), 0, m->levels - 1);
      int w = m_image_get_width(m->images[l]);
      int h = m_image_get_width(m->images[l]);
45    double i = fmod(fmod(w * x, w) + w, w);
      double j = h * y;
      int i0 = floor(i);
      int j0 = floor(j);
      int i1 = i0 + 1;
50    int j1 = j0 + 1;
      double ix = i - i0;
      double jx = j - j0;
      i0 = ((i0 % w) + w) % w;
      j0 = clamp(j0, 0, h - 1);
55    i1 = ((i1 % w) + w) % w;
      j1 = clamp(j1, 0, h - 1);
      m_pixel_t p00 = m_image_peek(m->images[l], i0, j0);
      m_pixel_t p10 = m_image_peek(m->images[l], i1, j0);
      m_pixel_t p01 = m_image_peek(m->images[l], i0, j1);
60    m_pixel_t p11 = m_image_peek(m->images[l], i1, j1);
      m_pixel_t p0 = m_pixel_mix(p00, p01, jx);
      m_pixel_t p1 = m_pixel_mix(p10, p11, jx);
      m_pixel_t p = m_pixel_mix(p0, p1, ix);
      return p;
65  }

    extern m_pixel_t m_mipmap_linear_linear(const m_mipmap *m, double level, double ↙
        ↳ x, double y) {
      int l0 = floor(level);
      int l1 = l0 + 1;
70    double lx = level - l0;
      return m_pixel_mix(m_mipmap_linear(m, l0, x, y), m_mipmap_linear(m, l1, x, y),↙
          ↳ lx);
    }

    extern int m_mipmap_get_levels(const m_mipmap *m) {
75    return m->levels;
    }
```

36 c/lib/m_pixel.c

```
#include <mandelbrot-graphics.h>
#include "m_d_util.h"

extern double m_pixel_red(m_pixel_t p) {
  return ((p & 0x00ff0000) >> 16) / (double) ((p & 0xff000000) >> 24);
}

extern double m_pixel_green(m_pixel_t p) {
  return ((p & 0x0000ff00) >> 8) / (double) ((p & 0xff000000) >> 24);
}

extern double m_pixel_blue(m_pixel_t p) {
  return ((p & 0x000000ff)) / (double) ((p & 0xff000000) >> 24);
}

extern double m_pixel_alpha(m_pixel_t p) {
  return ((p & 0xff000000) >> 24) / 255.0;
}

extern m_pixel_t m_pixel_rgba(double r, double g, double b, double a) {
  m_pixel_t ri = fmin(fmax(255 * r * a + 0.5, 0), 255);
  m_pixel_t gi = fmin(fmax(255 * g * a + 0.5, 0), 255);
  m_pixel_t bi = fmin(fmax(255 * b * a + 0.5, 0), 255);
  m_pixel_t ai = fmin(fmax(255 * a + 0.5, 0), 255);
  return (ai << 24) | (ri << 16) | (gi << 8) | bi;
}

extern m_pixel_t m_pixel_hsva(double h, double s, double v, double a) {
  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;
    }
  }
  return m_pixel_rgba(r, g, b, a);
}

extern m_pixel_t m_pixel_mix(m_pixel_t a, m_pixel_t b, double x) {
  return m_pixel_rgba
    ( mix(m_pixel_red(a), m_pixel_red(b), x)
    , mix(m_pixel_green(a), m_pixel_green(b), x)
    , mix(m_pixel_blue(a), m_pixel_blue(b), x)
    , mix(m_pixel_alpha(a), m_pixel_alpha(b), x)
```

```
    );
}

extern m_pixel_t m_pixel_blacken(m_pixel_t a, double x) {
  return m_pixel_mix(a, m_pixel_rgba(0, 0, 0, 1), x);
}

extern m_pixel_t m_pixel_whiten(m_pixel_t a, double x) {
  return m_pixel_mix(a, m_pixel_rgba(1, 1, 1, 1), x);
}
```

37 c/lib/pkgconfig/mandelbrot-graphics.pc.in

```
exec_prefix=${prefix}
libdir=${exec_prefix}/lib
includedir=${prefix}/include

Name: mandelbrot-graphics
Description: CPU-based rendering of the Mandelbrot set
Version: 0.1.0.0
URL: https://code.mathr.co.uk/mandelbrot-graphics
Requires: cairo mandelbrot-numerics
Libs: -fopenmp -L${libdir} -lmandelbrot-graphics
Libs.private: -lmpc -lmpfr -lgmp -lm
Cflags: -fopenmp -I${includedir}
```

38 COPYING

GNU GENERAL PUBLIC LICENSE
Version 3, 29 June 2007

Preamble

The GNU General Public License is a free, copyleft license for software and other kinds of works.

The licenses for most software and other practical works are designed to take away your freedom to share and change the works. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change all versions of a program--to make sure it remains free software for all its users. We, the Free Software Foundation, use the GNU General Public License for most of our software; it applies also to any other work released this way by its authors. You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for them if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs, and that you know you can do these things.

To protect your rights, we need to prevent others from denying you these rights or asking you to surrender the rights. Therefore, you have certain responsibilities if you distribute copies of the software, or if you modify it: responsibilities to respect the freedom of others.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must pass on to the recipients the same freedoms that you received. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

Developers that use the GNU GPL protect your rights with two steps: (1) assert copyright on the software, and (2) offer you this License giving you legal permission to copy, distribute and/or modify it.

For the developers' and authors' protection, the GPL clearly explains that there is no warranty for this free software. For both users' and authors' sake, the GPL requires that modified versions be marked as changed, so that their problems will not be attributed erroneously to authors of previous versions.

Some devices are designed to deny users access to install or run modified versions of the software inside them, although the manufacturer can do so. This is fundamentally incompatible with the aim of protecting users' freedom to change the software. The systematic pattern of such abuse occurs in the area of products for individuals to use, which is precisely where it is most unacceptable. Therefore, we have designed this version of the GPL to prohibit the practice for those products. If such problems arise substantially in other domains, we stand ready to extend this provision to those domains in future versions of the GPL, as needed to protect the freedom of users.

Finally, every program is threatened constantly by software patents. States should not allow patents to restrict development and use of software on general-purpose computers, but in those that do, we wish to avoid the special danger that patents applied to a free program could make it effectively proprietary. To prevent this, the GPL assures that patents cannot be used to render the program non-free.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS

0. Definitions.

"This License" refers to version 3 of the GNU General Public License.

"Copyright" also means copyright-like laws that apply to other kinds of works, such as semiconductor masks.

"The Program" refers to any copyrightable work licensed under this License. Each licensee is addressed as "you". "Licensees" and "recipients" may be individuals or organizations.

To "modify" a work means to copy from or adapt all or part of the work in a fashion requiring copyright permission, other than the making of an

exact copy. The resulting work is called a "modified version" of the earlier work or a work "based on" the earlier work.

A "covered work" means either the unmodified Program or a work based on the Program.

To "propagate" a work means to do anything with it that, without permission, would make you directly or secondarily liable for infringement under applicable copyright law, except executing it on a computer or modifying a private copy. Propagation includes copying, distribution (with or without modification), making available to the public, and in some countries other activities as well.

To "convey" a work means any kind of propagation that enables other parties to make or receive copies. Mere interaction with a user through a computer network, with no transfer of a copy, is not conveying.

An interactive user interface displays "Appropriate Legal Notices" to the extent that it includes a convenient and prominently visible feature that (1) displays an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically

linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice;

keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

a) The work must carry prominent notices stating that you modified it, and giving a relevant date.

b) The work must carry prominent notices stating that it is released under this License and any conditions added under section 7. This requirement modifies the requirement in section 4 to "keep intact all notices".

c) You must license the entire work, as a whole, under this License to anyone who comes into possession of a copy. This License will therefore apply, along with any applicable section 7 additional terms, to the whole of the work, and all its parts, regardless of how they are packaged. This License gives no permission to license the work in any other way, but it does not invalidate such permission if you have separately received it.

d) If the work has interactive user interfaces, each must display Appropriate Legal Notices; however, if the Program has interactive interfaces that do not display Appropriate Legal Notices, your work need not make them do so.

A compilation of a covered work with other separate and independent works, which are not by their nature extensions of the covered work, and which are not combined with it such as to form a larger program, in or on a volume of a storage or distribution medium, is called an "aggregate" if the compilation and its resulting copyright are not used to limit the access or legal rights of the compilation's users beyond what the individual works permit. Inclusion of a covered work in an aggregate does not cause this License to apply to the other parts of the aggregate.

6. Conveying Non-Source Forms.

You may convey a covered work in object code form under the terms of sections 4 and 5, provided that you also convey the machine-readable Corresponding Source under the terms of this License, in one of these ways:

a) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by the Corresponding Source fixed on a durable physical medium customarily used for software interchange.

b) Convey the object code in, or embodied in, a physical product (including a physical distribution medium), accompanied by a written offer, valid for at least three years and valid for as long as you offer spare parts or customer support for that product model, to give anyone who possesses the object code either (1) a copy of the Corresponding Source for all the software in the product that is covered by this License, on a durable physical medium customarily used for software interchange, for a price no more than your reasonable cost of physically performing this conveying of source, or (2) access to copy the Corresponding Source from a network server at no charge.

c) Convey individual copies of the object code with a copy of the written offer to provide the Corresponding Source. This alternative is allowed only occasionally and noncommercially, and only if you received the object code with such an offer, in accord with subsection 6b.

d) Convey the object code by offering access from a designated place (gratis or for a charge), and offer equivalent access to the Corresponding Source in the same way through the same place at no further charge. You need not require recipients to copy the Corresponding Source along with the object code. If the place to copy the object code is a network server, the Corresponding Source may be on a different server (operated by you or a third party) that supports equivalent copying facilities, provided you maintain clear directions next to the object code saying where to find the Corresponding Source. Regardless of what server hosts the Corresponding Source, you remain obligated to ensure that it is available for as long as needed to satisfy these requirements.

e) Convey the object code using peer-to-peer transmission, provided you inform other peers where the object code and Corresponding Source of the work are being offered to the general public at no charge under subsection 6d.

A separable portion of the object code, whose source code is excluded from the Corresponding Source as a System Library, need not be included in conveying the object code work.

A "User Product" is either (1) a "consumer product", which means any tangible personal property which is normally used for personal, family, or household purposes, or (2) anything designed or sold for incorporation into a dwelling. In determining whether a product is a consumer product, doubtful cases shall be resolved in favor of coverage. For a particular product received by a particular user, "normally used" refers to a typical or common use of that class of product, regardless of the status of the particular user or of the way in which the particular user actually uses, or expects or is expected to use, the product. A product is a consumer product regardless of whether the product has substantial commercial, industrial or non-consumer uses, unless such uses represent the only significant mode of use of the product.

"Installation Information" for a User Product means any methods, procedures, authorization keys, or other information required to install and execute modified versions of a covered work in that User Product from a modified version of its Corresponding Source. The information must

suffice to ensure that the continued functioning of the modified object code is in no case prevented or interfered with solely because modification has been made.

If you convey an object code work under this section in, or with, or specifically for use in, a User Product, and the conveying occurs as part of a transaction in which the right of possession and use of the User Product is transferred to the recipient in perpetuity or for a fixed term (regardless of how the transaction is characterized), the Corresponding Source conveyed under this section must be accompanied by the Installation Information. But this requirement does not apply if neither you nor any third party retains the ability to install modified object code on the User Product (for example, the work has been installed in ROM).

The requirement to provide Installation Information does not include a requirement to continue to provide support service, warranty, or updates for a work that has been modified or installed by the recipient, or for the User Product in which it has been modified or installed. Access to a network may be denied when the modification itself materially and adversely affects the operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or

b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or

c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or

d) Limiting the use for publicity purposes of names of licensors or authors of the material; or

e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or

f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a license from the original licensors, to run, modify and propagate that work, subject to this License. You are not responsible for enforcing compliance by third parties with this License.

An "entity transaction" is a transaction transferring control of an organization, or substantially all assets of one, or subdividing an organization, or merging organizations. If propagation of a covered work results from an entity transaction, each party to that transaction who receives a copy of the work also receives whatever licenses to the work the party's predecessor in interest had or could give under the previous paragraph, plus a right to possession of the Corresponding Source of the work from the predecessor in interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights granted or affirmed under this License. For example, you may not impose a license fee, royalty, or other charge for exercise of rights granted under this License, and you may not initiate litigation (including a cross-claim or counterclaim in a lawsuit) alleging that any patent claim is infringed by making, using, selling, offering for sale, or importing the Program or any portion of it.

11. Patents.

A "contributor" is a copyright holder who authorizes use under this License of the Program or a work on which the Program is based. The work thus licensed is called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or controlled by the contributor, whether already acquired or hereafter acquired, that would be infringed by some manner, permitted by this License, of making, using, or selling its contributor version, but do not include claims that would be infringed only as a consequence of further modification of the contributor version. For purposes of this definition, "control" includes the right to grant patent sublicenses in a manner consistent with the requirements of

this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent license under the contributor's essential patent claims, to make, use, sell, offer for sale, import and otherwise run, modify and propagate the contents of its contributor version.

In the following three paragraphs, a "patent license" is any express agreement or commitment, however denominated, not to enforce a patent (such as an express permission to practice a patent or covenant not to sue for patent infringement). To "grant" such a patent license to a party means to make such an agreement or commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the Corresponding Source of the work is not available for anyone to copy, free of charge and under the terms of this License, through a publicly available network server or other readily accessible means, then you must either (1) cause the Corresponding Source to be so available, or (2) arrange to deprive yourself of the benefit of the patent license for this particular work, or (3) arrange, in a manner consistent with the requirements of this License, to extend the patent license to downstream recipients. "Knowingly relying" means you have actual knowledge that, but for the patent license, your conveying the covered work in a country, or your recipient's use of the covered work in a country, would infringe one or more identifiable patents in that country that you have reason to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you convey, or propagate by procuring conveyance of, a covered work, and grant a patent license to some of the parties receiving the covered work authorizing them to use, propagate, modify or convey a specific copy of the covered work, then the patent license you grant is automatically extended to all recipients of the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of its coverage, prohibits the exercise of, or is conditioned on the non-exercise of one or more of the rights that are specifically granted under this License. You may not convey a covered work if you are a party to an arrangement with a third party that is in the business of distributing software, under which you make payment to the third party based on the extent of your activity of conveying the work, and under which the third party grants, to any of the parties who would receive the covered work from you, a discriminatory patent license (a) in connection with copies of the covered work conveyed by you (or copies made from those copies), or (b) primarily for and in connection with specific products or compilations that contain the covered work, unless you entered into that arrangement, or that patent license was granted, prior to 28 March 2007.

Nothing in this License shall be construed as excluding or limiting any implied license or other defenses to infringement that may otherwise be available to you under applicable patent law.

12. No Surrender of Others' Freedom.

If conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot convey a covered work so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not convey it at all. For example, if you agree to terms that obligate you to collect a royalty for further conveying from those to whom you convey the Program, the only way you could satisfy both those terms and this License would be to refrain entirely from conveying the Program.

13. Use with the GNU Affero General Public License.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU Affero General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the special requirements of the GNU Affero General Public License, section 13, concerning interaction through a network will apply to the combination as such.

14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU General Public License can be used, that proxy's public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. Limitation of Liability.

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

17. Interpretation of Sections 15 and 16.

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Program, unless a warranty or assumption of liability accompanies a copy of the Program in return for a fee.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively state the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

> <one line to give the program's name and a brief idea of what it does.>
> Copyright (C) <year> <name of author>
>
> This program is free software: you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.
>
> This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.
>
> You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/>.

Also add information on how to contact you by electronic and paper mail.

If the program does terminal interaction, make it output a short notice like this when it starts in an interactive mode:

> <program> Copyright (C) <year> <name of author>

```
    This program comes with ABSOLUTELY NO WARRANTY; for details type 'show w'.
    This is free software, and you are welcome to redistribute it
    under certain conditions; type 'show c' for details.

The hypothetical commands 'show w' and 'show c' should show the appropriate
parts of the General Public License.  Of course, your program's commands
might be different; for a GUI interface, you would use an "about box".

  You should also get your employer (if you work as a programmer) or school,
if any, to sign a "copyright disclaimer" for the program, if necessary.
For more information on this, and how to apply and follow the GNU GPL, see
<http://www.gnu.org/licenses/>.

  The GNU General Public License does not permit incorporating your program
into proprietary programs.  If your program is a subroutine library, you
may consider it more useful to permit linking proprietary applications with
the library.  If this is what you want to do, use the GNU Lesser General
Public License instead of this License.  But first, please read
<http://www.gnu.org/philosophy/why-not-lgpl.html>.
```

39 .gitignore

```
*.a
*.d
*.o
*.pc
*.so
*.png
c/bin/m-cardioid-warping
c/bin/m-dense-misiurewicz
c/bin/m-feigenbaum-zoom
c/bin/m-fibonacci-phi-zoom
c/bin/m-misiurewicz-domains
c/bin/m-period-scan
c/bin/m-stretching-cusps
c/bin/m-subwake-diagram-a
c/bin/m-subwake-diagram-b
c/bin/m-subwake-diagram-c
c/bin/m-render
c/bin/m-average-distance
c/bin/m-buddhabrot-convergence
c/bin/m-distorted-minibrot
c/bin/m-fibonacci-phi-animation
c/bin/m-homunculus
c/bin/m-island-zoom
c/bin/m-render-elephant-valley
c/bin/m-subwake-diagram
c/bin/m-warped-midgets
c/bin/m-misiurewicz-basins
c/bin/m-nucleus-basins
c/bin/m-furcation-rainbow
c/bin/m-algorithm-9
c/bin/m-stretching-feigenbaum
.cabal-sandbox/
cabal.sandbox.config
dist
_
```

```
c/bin/m-misiurewicz-domains-2
c/bin/m-mu-unit
c/bin/m-render-julia
```

40 hs/bin/m-trustworthy-julia.hs

```
module Main (main) where

import Control.Monad (forM_)
import Data.Foldable (toList)
import Data.Maybe (mapMaybe)
import System.Environment (getArgs)
import qualified Data.Set as S

import Mandelbrot.Graphics.Trustworthy.Julia
import Mandelbrot.Graphics.Trustworthy.Quadratic
import qualified Mandelbrot.Graphics.BoundingBox.D1 as D1
import Mandelbrot.Graphics.BoundingBox.D2
import Mandelbrot.Graphics.QuadTree

main :: IO ()
main = do
  args <- getArgs
  case args of
    [outfile, sre, sim, sdepth] -> do
      re <- readIO sre
      im <- readIO sim
      depth <- readIO sdepth
      let r :: Double
          r = max (sqrt $ re^2 + im^2) 2
          images = julia (quadratic (singleton re im)) r
          content = svg r (images !! depth)
      writeFile outfile content

svg :: Real r => r -> Image r -> String
{-# SPECIALIZE svg :: Double -> Image Double -> String #-}
{-# SPECIALIZE svg :: Rational -> Image Rational -> String #-}
svg r i = unlines $
  [ "<?xml version='1.0' encoding='UTF-8' ?>"
  , "<svg xmlns='http://www.w3.org/2000/svg' version='1.0' viewBox='" ++ f (-r) ↙
    ↳ ++ " " ++ f (-r) ++ " " ++ f (2 * r) ++ " " ++ f (2 * r) ++ "' width↙
    ↳ ='4096' height='4096'>"
  , "<g stroke='none'>"
  ] ++ map (uncurry s) (toList i) ++
  [ "</g>"
  , "</svg>"
  ]
  where
    f z = show (realToFrac z :: Double)
    s b c = svgBox b (rgb c)
    rgb Black  = "black"
    rgb Gray   = "red"
    rgb White  = "white"
    rgb Marked = "yellow"

svgBox :: Real r => Box r -> String -> String
{-# SPECIALIZE svgBox :: Box Double -> String -> String #-}
```

```
{-# SPECIALIZE svgBox :: Box Rational -> String -> String #-}
svgBox (Box (D1.Box xlo xhi) (D1.Box ylo yhi)) fill = unwords $
      [ "<path d='M"
      , x0, ",", y0, "L"
      , x0, ",", y1, "L"
      , x1, ",", y1, "L"
      , x1, ",", y0, "L"
      , x0, ",", y0, "Z' fill='" ++ fill ++ "' />"
      ]
      where
        [x0, y0, x1, y1] = map f [xlo, ylo, xhi, yhi]
        f z = show (realToFrac z :: Double)
```

41 hs/bin/m-trustworthy-mandelbrot.hs

```
module Main (main) where

import Control.Monad (forM_)
import Data.Foldable (toList)
import Data.Maybe (mapMaybe)
import System.Environment (getArgs)
import qualified Data.Set as S

import Mandelbrot.Graphics.Trustworthy.Mandelbrot
import Mandelbrot.Graphics.Trustworthy.Quadratic
import qualified Mandelbrot.Graphics.BoundingBox.D1 as D1
import Mandelbrot.Graphics.BoundingBox.D2
import Mandelbrot.Graphics.QuadTree

main :: IO ()
main = do
  let r :: Double
      r = 2
      images = mandelbrot 0 (symmetric r)
      content = map (svg r) images
  forM_ (zip [0..] content) $ \(d, s) -> do
    let filename = "M" ++ show2 d ++ ".svg"
    writeFile filename s

svg :: Real r => r -> Image r -> String
{-# SPECIALIZE svg :: Double -> Image Double -> String #-}
{-# SPECIALIZE svg :: Rational -> Image Rational -> String #-}
svg r i = unlines $
  [ "<?xml version='1.0' encoding='UTF-8' ?>"
  , "<svg xmlns='http://www.w3.org/2000/svg' version='1.0' viewBox='" ++ f (-r) ↙
    ↳ ++ " " ++ f (-r) ++ " " ++ f (2 * r) ++ " " ++ f (2 * r) ++ "' width↙
    ↳ ='4096' height='4096'>"
  , "<g stroke='grey' stroke-width='0.001'>"
  ] ++ map (uncurry s) (toList i) ++
  [ "</g>"
  , "</svg>"
  ]
  where
    f z = show (realToFrac z :: Double)
    s b c = svgBox b (rgb c)
    rgb Black  = "black"
    rgb Gray   = "red"
```

```
    rgb White  = "white"
    rgb Marked = "yellow"

svgBox :: Real r => Box r -> String -> String
{-# SPECIALIZE svgBox :: Box Double -> String -> String #-}
{-# SPECIALIZE svgBox :: Box Rational -> String -> String #-}
svgBox (Box (D1.Box xlo xhi) (D1.Box ylo yhi)) fill = unwords $
      [ "<path d='M"
      , x0, ",", y0, "L"
      , x0, ",", y1, "L"
      , x1, ",", y1, "L"
      , x1, ",", y0, "L"
      , x0, ",", y0, "Z' fill='" ++ fill ++ "' />"
      ]
      where
        [x0, y0, x1, y1] = map f [xlo, ylo, xhi, yhi]
        f z = show (realToFrac z :: Double)

show2 :: Int -> String
show2 n
  | n < 10 = "0" ++ show n
  | otherwise = show n
```

42 hs/lib/Mandelbrot/Graphics/BoundingBox/D1.hs

```
{-# LANGUAGE DeriveFunctor #-}
{-# LANGUAGE DeriveFoldable #-}
{-# LANGUAGE DeriveTraversable #-}
module Mandelbrot.Graphics.BoundingBox.D1 where

data Box r = Box
  { lower :: !r
  , upper :: !r
  }
  deriving (Eq, Read, Show, Functor, Foldable, Traversable)

box :: Ord r => r -> r -> Box r
{-# SPECIALIZE box :: Double -> Double -> Box Double #-}
{-# SPECIALIZE box :: Rational -> Rational -> Box Rational #-}
box a b
  | a <= b    = Box a b
  | otherwise = Box b a

midPoint :: Fractional r => Box r -> r
{-# SPECIALIZE midPoint :: Box Double -> Double #-}
{-# SPECIALIZE midPoint :: Box Rational -> Rational #-}
midPoint (Box a b) = (a + b) / 2

split :: RealFrac r => Box r -> (Box r, Box r)
{-# SPECIALIZE split :: Box Double -> (Box Double, Box Double) #-}
{-# SPECIALIZE split :: Box Rational -> (Box Rational, Box Rational) #-}
split x@(Box a b) = (Box a m, Box m b) where m = midPoint x

inside  :: Ord r => Box r -> Box r -> Bool
{-# SPECIALIZE inside :: Box Double -> Box Double -> Bool #-}
{-# SPECIALIZE inside :: Box Rational -> Box Rational -> Bool #-}
inside  (Box xlo xhi) (Box ulo uhi) = ulo <= xlo && xhi <= uhi
```

```
inside' :: Ord r => Box r -> Box r -> Bool
{-# SPECIALIZE inside' :: Box Double -> Box Double -> Bool #-}
{-# SPECIALIZE inside' :: Box Rational -> Box Rational -> Bool #-}
inside' (Box xlo xhi) (Box ulo uhi) = ulo <  xlo && xhi <  uhi

disjoint  :: Ord r => Box r -> Box r -> Bool
{-# SPECIALIZE disjoint :: Box Double -> Box Double -> Bool #-}
{-# SPECIALIZE disjoint :: Box Rational -> Box Rational -> Bool #-}
disjoint  (Box xlo xhi) (Box ulo uhi) = xlo >= uhi || xhi <= ulo

disjoint' :: Ord r => Box r -> Box r -> Bool
{-# SPECIALIZE disjoint' :: Box Double -> Box Double -> Bool #-}
{-# SPECIALIZE disjoint' :: Box Rational -> Box Rational -> Bool #-}
disjoint' (Box xlo xhi) (Box ulo uhi) = xlo >  uhi || xhi <  ulo

symmetric :: Num r => r -> Box r
{-# SPECIALIZE symmetric :: Double -> Box Double #-}
{-# SPECIALIZE symmetric :: Rational -> Box Rational #-}
symmetric r = Box (negate r') r' where r' = abs r

singleton :: r -> Box r
{-# SPECIALIZE singleton :: Double -> Box Double #-}
{-# SPECIALIZE singleton :: Rational -> Box Rational #-}
singleton r = Box r r

union :: Ord r => Box r -> Box r -> Box r
{-# SPECIALIZE union :: Box Double -> Box Double -> Box Double #-}
{-# SPECIALIZE union :: Box Rational -> Box Rational -> Box Rational #-}
union (Box a b) (Box u v) = Box (min a u) (max b v)
```

43 hs/lib/Mandelbrot/Graphics/BoundingBox/D2.hs

```
{-# LANGUAGE DeriveFunctor #-}
{-# LANGUAGE DeriveFoldable #-}
{-# LANGUAGE DeriveTraversable #-}
module Mandelbrot.Graphics.BoundingBox.D2 where

import qualified Mandelbrot.Graphics.BoundingBox.D1 as D1

data Box r = Box
  { xRange :: !(D1.Box r)
  , yRange :: !(D1.Box r)
  }
  deriving (Eq, Read, Show, Functor, Foldable, Traversable)

split :: RealFrac r => Box r -> (Box r, Box r, Box r, Box r)
{-# SPECIALIZE split :: Box Double -> (Box Double, Box Double, Box Double, Box ↙
    ↳ Double) #-}
{-# SPECIALIZE split :: Box Rational -> (Box Rational, Box Rational, Box ↙
    ↳ Rational, Box Rational) #-}
split (Box x y) = (Box xlo ylo, Box xlo yhi, Box xhi ylo, Box xhi yhi)
  where
    (xlo, xhi) = D1.split x
    (ylo, yhi) = D1.split y

inside  :: Ord r => Box r -> Box r -> Bool
```

```
{-# SPECIALIZE inside :: Box Double -> Box Double -> Bool #-}
{-# SPECIALIZE inside :: Box Rational -> Box Rational -> Bool #-}
inside (Box x y) (Box u v) = x `D1.inside` u && y `D1.inside` v

inside' :: Ord r => Box r -> Box r -> Bool
{-# SPECIALIZE inside' :: Box Double -> Box Double -> Bool #-}
{-# SPECIALIZE inside' :: Box Rational -> Box Rational -> Bool #-}
inside' (Box x y) (Box u v) = x `D1.inside'` u && y `D1.inside'` v

disjoint :: Ord r => Box r -> Box r -> Bool
{-# SPECIALIZE disjoint :: Box Double -> Box Double -> Bool #-}
{-# SPECIALIZE disjoint :: Box Rational -> Box Rational -> Bool #-}
disjoint (Box x y) (Box u v) = x `D1.disjoint` u || y `D1.disjoint` v

disjoint' :: Ord r => Box r -> Box r -> Bool
{-# SPECIALIZE disjoint' :: Box Double -> Box Double -> Bool #-}
{-# SPECIALIZE disjoint' :: Box Rational -> Box Rational -> Bool #-}
disjoint' (Box x y) (Box u v) = x `D1.disjoint'` u || y `D1.disjoint'` v

symmetric :: Num r => r -> Box r
{-# SPECIALIZE symmetric :: Double -> Box Double #-}
{-# SPECIALIZE symmetric :: Rational -> Box Rational #-}
symmetric r = Box b b where b = D1.symmetric r

singleton :: r -> r -> Box r
{-# SPECIALIZE singleton :: Double -> Double -> Box Double #-}
{-# SPECIALIZE singleton :: Rational -> Rational -> Box Rational #-}
singleton x y = Box (D1.singleton x) (D1.singleton y)

union :: Ord r => Box r -> Box r -> Box r
{-# SPECIALIZE union :: Box Double -> Box Double -> Box Double #-}
{-# SPECIALIZE union :: Box Rational -> Box Rational -> Box Rational #-}
union (Box a b) (Box u v) = Box (D1.union a u) (D1.union b v)
```

44 hs/lib/Mandelbrot/Graphics/Colour.hs

```
module Mandelbrot.Graphics.Colour
  ( monochrome
  ) where

import Data.Word (Word8)

import Mandelbrot.Graphics.Compute (Compute(..))

monochrome :: RealFloat r => Compute r -> Word8
monochrome (Exterior{ computeDE = de }) = round (255 * tanh de)
monochrome (Interior{ computeDE = de }) = round (255 * tanh de)
monochrome _ = 0
```

45 hs/lib/Mandelbrot/Graphics/Compute.hs

```
{-# LANGUAGE BangPatterns #-}
module Mandelbrot.Graphics.Compute
  ( Compute(..)
  , Partial(..)
  , compute
```

```
    , initialize
    , step
    , fixup
    , isInterior
    , interiorDE
    ) where

import Data.Complex (Complex((:+)), magnitude)
import Data.Maybe (isNothing, isJust)

import Mandelbrot.Numerics (Square, Approx, wucleus)

norm :: Num r => Complex r -> r
norm (x :+ y) = x * x + y * y

data Partial r = Partial !Int !(Complex r)

data Compute r
  = Interior{ computeP :: !Int, computeDZ :: !(Complex r), computeDE :: !r }
  | Exterior{ computeP, computeN :: !Int, computeZ :: !(Complex r), computeDE :: !r }
  | Unknown{ computePartials :: Maybe [Partial r], computeP, computeN :: !Int, computeC, computeDC0, computeZ, computeDC :: !(Complex r), computeER2, computeMZ2 :: !r }

initialize :: RealFloat r => Bool -> r -> (Complex r, Complex r) -> Compute r
initialize bias er (c, dc) = Unknown
  { computePartials = if bias then Nothing else Just []
  , computeP = 0
  , computeN = 0
  , computeC = c
  , computeDC0 = dc
  , computeZ = 0
  , computeDC = 0
  , computeER2 = er * er
  , computeMZ2 = 1 / 0
  }

step :: (RealFloat r, Square r, Approx r) => Compute r -> Compute r
step u@Unknown{ computePartials = mps, computeN = n, computeC = c, computeDC0 = dc00, computeZ = z0, computeDC = dc0, computeER2 = er2, computeMZ2 = mz2 }
  | z2 < mz2 && isNothing mps && isJust interior = case interior of
      Just (de, dz) -> Interior{ computeP = p, computeDZ = dz, computeDE = de / magnitude dc00 }
  | z2 < mz2 && z2 < er2 = u{ computePartials = (Partial p z :) `fmap` mps, computeP = p, computeN = p, computeZ = z, computeDC = dc, computeMZ2 = z2 }
  | z2 < er2 = u{ computeN = p, computeZ = z, computeDC = dc }
  | otherwise = Exterior{ computeP = computeP u, computeN = p, computeZ = z, computeDE = exterior }
  where
    dc = 2 * z0 * dc0 + dc00
    z = z0 * z0 + c
    z2 = norm z
    p = n + 1
    interior = interiorDE z c p 64
    exterior = 2 * magnitude z * log(magnitude z) / magnitude dc
```

```
55  step u = u

    fixup :: (RealFloat r, Square r, Approx r) => Compute r -> Compute r
    fixup u@Unknown{ computePartials = Just ps, computeC = c } = go (reverse ps)
      where
60      go [] = u{ computePartials = Just [] }
        go (Partial p z : pzs) = case interiorDE z c p 64 of
          Just (de, dz) -> Interior{ computeP = p, computeDZ = dz, computeDE = de }
          Nothing -> go pzs
    fixup u = u
65
    compute :: (RealFloat r, Square r, Approx r) => Bool -> r -> (Complex r, Complex↙
        ↳ r) -> Int -> Compute r
    compute bias er cdc = go (initialize bias er cdc)
      where
        go r@Exterior{} _ = r
70      go r@Interior{} _ = r
        go r 0 = if bias then r else fixup r
        go r n = go (step r) (n - 1)

    isInterior :: Compute r -> Bool
75  isInterior Interior{} = True
    isInterior _ = False

    interiorDE :: (RealFloat r, Square r, Approx r) => Complex r -> Complex r -> Int↙
        ↳ -> Int -> Maybe (r, Complex r)
    interiorDE z0 c p steps = case last . take steps . (z0 :) $ wucleus p c z0 of
80    z00 -> case go1 p z00 1 of
        dz0 | norm dz0 <= 1 -> Just (go2 p z00 1 0 0 0)
        _ -> Nothing
      where
        go1 q !z !dz
85        | q > 0 = go1 (q - 1) (z * z + c) (2 * z * dz)
          | otherwise = dz
        go2 q !z !dz !dzdz !dc !dcdz
          | q > 0 = go2 (q - 1) (z * z + c) (2 * z * dz) (2 * (dz * dz + z * dzdz)) ↙
              ↳ (2 * z * dc + 1) (2 * (z * dcdz + dz * dc))
          | otherwise = ((1 - norm dz) / magnitude (dcdz + dzdz * dc / (1 - dz)), dz↙
              ↳ )
```

46 hs/lib/Mandelbrot/Graphics.hs

```
    module Mandelbrot.Graphics
      ( module Mandelbrot.Graphics.Colour
      , module Mandelbrot.Graphics.Compute
      , module Mandelbrot.Graphics.Matrix
5     , module Mandelbrot.Graphics.Render
      , module Mandelbrot.Graphics.Transform
      ) where

    import Mandelbrot.Graphics.Colour
10  import Mandelbrot.Graphics.Compute
    import Mandelbrot.Graphics.Matrix hiding (moebius3)
    import Mandelbrot.Graphics.Render
    import Mandelbrot.Graphics.Transform
```

47 hs/lib/Mandelbrot/Graphics/Matrix.hs

```haskell
module Mandelbrot.Graphics.Matrix
  ( Mat2(..)
  , identity
  , det
  , tr
  , inv
  , mul
  , diagonalize
  , moebius3
  , interpolate
  ) where

data Mat2 r = Mat2 !r !r !r !r

identity :: Num r => Mat2 r
identity = Mat2 1 0 0 1

det, tr :: Num r => Mat2 r -> r
det (Mat2 a b c d) = a * d - b * c
tr (Mat2 a _ _ d) = a + d

inv :: Fractional r => Mat2 r -> Mat2 r
inv m@(Mat2 a b c d) =
  let e = det m
  in  Mat2 (d / e) (-b/e) (-c/e) (a/e)

mul :: Num r => Mat2 r -> Mat2 r -> Mat2 r
mul (Mat2 la lb lc ld) (Mat2 ra rb rc rd) = Mat2 a b c d
  where
    a = la * ra + lb * rc
    b = la * rb + lb * rd
    c = lc * ra + ld * rc
    d = lc * rb + ld * rd

diagonalize :: (Eq r, Floating r) => Mat2 r -> (Mat2 r, Mat2 r, Mat2 r)
diagonalize m@(Mat2 ma mb mc md) = (p, d, p1)
  where
    d = Mat2 l1 0 0 l2
    l1 = tr2 + k
    l2 = tr2 - k
    k = sqrt (tr2 * tr2 - det m)
    tr2 = tr m / 2
    p1 = inv p
    p | mb /= 0 = Mat2 mb mb (l1 - ma) (l2 - ma)
      | mc /= 0 = Mat2 (l1 - md) (l2 - md) mc mc
      | otherwise = identity

moebius3 :: Num r => r -> r -> r -> Mat2 r
moebius3 zero one infinity = Mat2 a b c d
  where
    a = infinity * (zero - one)
    b = zero * (one - infinity)
    c = zero - one
    d = one - infinity

```

```
interpolate :: (Eq r, Floating r) => Mat2 r -> Mat2 r -> r -> Mat2 r
interpolate f g = \t ->
    let e = Mat2 (da ** t) 0 0 (dd ** t)
        fpe = mul fp e
    in  mul fpe p1
  where
    f1 = inv f
    f1g = mul f1 g
    (p, Mat2 da _ _ dd, p1) = diagonalize f1g
    fp = mul f p
```

48 hs/lib/Mandelbrot/Graphics/QuadTree.hs

```
{-# LANGUAGE BangPatterns #-}
{-# LANGUAGE DeriveFunctor #-}
{-# LANGUAGE DeriveFoldable #-}
{-# LANGUAGE DeriveTraversable #-}
module Mandelbrot.Graphics.QuadTree where

import Mandelbrot.Graphics.BoundingBox.D2

data Tree a = Leaf a | Quad (Tree a) (Tree a) (Tree a) (Tree a)
  deriving (Eq, Ord, Read, Show, Functor, Foldable, Traversable)

boxed :: RealFrac r => Box r -> Tree a -> Tree (Box r, a)
{-# SPECIALIZE boxed :: Box Double -> Tree a -> Tree (Box Double, a) #-}
{-# SPECIALIZE boxed :: Box Rational -> Tree a -> Tree (Box Rational, a) #-}
boxed !z (Leaf a) = Leaf (z, a)
boxed !z (Quad a b c d) =
    Quad (boxed za a) (boxed zb b) (boxed zc c) (boxed zd d)
  where
    (za, zb, zc, zd) = split z

intersection :: RealFrac r => Box r -> Tree a -> Box r -> [a]
{-# SPECIALIZE intersection :: Box Double -> Tree a -> Box Double -> [a] #-}
{-# SPECIALIZE intersection :: Box Rational -> Tree a -> Box Rational -> [a] #-}
intersection bounds tree target = go bounds tree
  where
    go z t
      | target `disjoint` z = []
      | otherwise = case t of
          Leaf l -> [l]
          Quad a b c d ->
            let (za, zb, zc, zd) = split z
            in  concat
                  [ go za a
                  , go zb b
                  , go zc c
                  , go zd d
                  ]
```

49 hs/lib/Mandelbrot/Graphics/Render.hs

```
module Mandelbrot.Graphics.Render
  ( scanline
  , scanline1
```

```
  ) where

import Data.Complex (Complex((:+)))
import Data.List (foldl')

import Mandelbrot.Numerics (Square, Approx)

import Mandelbrot.Graphics.Transform (Transform(..))
import Mandelbrot.Graphics.Compute (Compute, compute, isInterior)

scanline1 :: (RealFloat r, Square r, Approx r) => Int -> Transform r -> r -> Int↙
    ↳ -> Int -> [Compute r]
scanline1 width transform er maxiters y = snd $ foldl' go (False, []) [width-1, ↙
    ↳ width-2 .. 0]
  where
    go (bias, acc) x =
      let px = compute bias er (forward transform (fromIntegral x :+ ↙
          ↳ fromIntegral y, 1)) maxiters
      in  (isInterior px, px : acc)

scanline :: (RealFloat r, Square r, Approx r) => Int -> Int -> Transform r -> r ↙
    ↳ -> Int -> [Compute r]
scanline width height transform er maxiters = concatMap (scanline1 width ↙
    ↳ transform er maxiters) [0 .. height - 1]
```

50 hs/lib/Mandelbrot/Graphics/STRef/Lazy.hs

```
{-# LANGUAGE RankNTypes #-}
{-# LANGUAGE GeneralizedNewtypeDeriving #-}
module Mandelbrot.Graphics.STRef.Lazy
  ( Ref()
  , RefM()
  , run
  , new
  , read
  , write
  , modify
  , st
  )
  where

import Prelude hiding (read)

import Control.Monad.Fix (MonadFix(..))
import Control.Monad.Fail (MonadFail(..))
import Control.Monad.State (StateT, evalStateT, get, put, lift)
import Control.Monad.ST.Lazy (ST, runST)
import Data.STRef.Lazy (STRef, newSTRef, readSTRef, writeSTRef, modifySTRef)
import Numeric.Natural (Natural)

data Ref s a = Ref !Natural !(STRef s a)
instance Eq (Ref s a) where Ref _ i == Ref _ j = i == j
instance Ord (Ref s a) where Ref i _ `compare` Ref j _ = i `compare` j

newtype RefM s a = RefM{ runRefM :: StateT Natural (ST s) a }
  deriving (Functor, Applicative, Monad, MonadFix, MonadFail)

```

```
run :: (forall s . RefM s a) -> a
run a = runST (evalStateT (runRefM a) 0)

next :: RefM s Natural
next = RefM $ do
  i <- get
  put $! succ i
  pure i

new :: a -> RefM s (Ref s a)
new a = Ref <$> next <*> st (newSTRef a)

read :: Ref s a -> RefM s a
read (Ref _ r) = st (readSTRef r)

write :: Ref s a -> a -> RefM s ()
write (Ref _ r) a = st (writeSTRef r a)

modify :: Ref s a -> (a -> a) -> RefM s ()
modify (Ref _ r) f = st (modifySTRef r f)

st :: ST s a -> RefM s a
st a = RefM (lift a)
```

51 hs/lib/Mandelbrot/Graphics/Transform.hs

```
module Mandelbrot.Graphics.Transform
  ( Transform(..)
  , invert
  , compose
  , rectangular
  , linear
  , moebius
  , moebius3
  , moebius6
  , cardioid
  ) where

import Prelude hiding (reverse)

import Data.Complex (Complex((:+)), conjugate)

import Mandelbrot.Graphics.Matrix (Mat2(..))
import qualified Mandelbrot.Graphics.Matrix as M

data Transform r = Transform
  { forward, reverse :: (Complex r, Complex r) -> (Complex r, Complex r) }

invert :: Transform r -> Transform r
invert (Transform f r) = Transform r f

compose :: Transform r -> Transform r -> Transform r
compose (Transform a a') (Transform b b') = Transform (b . a) (a' . b')

rectangular :: RealFloat r => Int -> Int -> Complex r -> r -> Transform r
rectangular width height c r = Transform
  { forward = \(c0, dc0) ->
```

```
      ( c + ((r / h2) :+ 0) * conjugate (c0 - ((w2 - 0.5) :+ (h2 - 0.5)))
      , conjugate ((r / (fromIntegral height / 2) :+ 0) * dc0)
      )
    , reverse = \(c0, dc0) ->
      ( conjugate ((c0 - c) * ((h2 / r) :+ 0)) + ((w2 - 0.5) :+ (h2 - 0.5))
      , conjugate (((h2 / r) :+ 0) * dc0)
      )
  }
  where
    h2 = fromIntegral height / 2
    w2 = fromIntegral width / 2

linear :: RealFloat r => Complex r -> Complex r -> Transform r
linear mul add = Transform
  { forward = \(c0, dc0) -> (mul * c0 + add, mul * dc0)
  , reverse = \(c0, dc0) -> ((c0 - add) / mul, dc0 / mul)
  }

moebius :: RealFloat r => Mat2 (Complex r) -> Transform r
moebius m@(Mat2 a b c d) = Transform
  { forward = \(c0, dc0) ->
      let e = c * c0 + d
      in  ((a * c0 + b) / e, dc0 * M.det m / (e * e))
  , reverse = \(c0, dc0) ->
      let e = a - c * c0
      in  ((d * c0 - b) / e, dc0 * M.det m / (e * e))
  }

moebius3 :: RealFloat r => Complex r -> Complex r -> Complex r -> Transform r
moebius3 zero one infinity = moebius (M.moebius3 zero one infinity)

moebius6 :: RealFloat r => Complex r -> Complex r -> Complex r -> Complex r -> ↙
    ↳ Complex r -> Complex r -> Transform r
moebius6 a0 a1 a2 b0 b1 b2 = moebius (M.mul (M.moebius3 a0 a1 a2) (M.inv (M.↙
    ↳ moebius3 b0 b1 b2)))

cardioid :: RealFloat r => Transform r
cardioid = Transform
  { forward = \(c0, dc0) ->
      let r = sqrt (1 - 4 * c0)
      in  (c0 - 1, dc0 * (-2) / r)
  , reverse = \(c0, dc0) ->
      let r = c0 + 1
      in  ((1 - r * r) / 4, dc0 * r / (-2))
  }
```

52 hs/lib/Mandelbrot/Graphics/Trustworthy/Julia.hs

```
{-
    An independent implementation of the algorithm in:

    "Images of Julia sets that you can trust"
    L. H. de Figueiredo, D. Nehab, J. Stolfi, and J. B. Oliveira
    Last updated on January 8, 2013 at 10:45am.
    <http://webdoc.sub.gwdg.de/ebook/serien/e/IMPA_A/721.pdf>
-}
module Mandelbrot.Graphics.Trustworthy.Julia
```

```
    ( Color(..)
    , Image(..)
    , julia
    ) where

import Control.Monad (forM_, when)
import Control.Monad.Loops (allM, anyM)
import Data.Ix (Ix(..))
import Data.Map (Map)
import qualified Data.Map as M
import Data.Set (Set)
import qualified Data.Set as S

import Mandelbrot.Graphics.STRef.Lazy (Ref, RefM)
import qualified Mandelbrot.Graphics.STRef.Lazy as R

import Mandelbrot.Graphics.BoundingBox.D2 (Box, inside', disjoint')
import qualified Mandelbrot.Graphics.BoundingBox.D2 as B

import Mandelbrot.Graphics.QuadTree hiding (Tree, intersection)
import qualified Mandelbrot.Graphics.QuadTree as Q

data Color = Black | Gray | White | Marked
  deriving (Eq, Ord, Read, Show, Enum, Bounded, Ix)

type Cell s = Ref s Color
type Tree s = Q.Tree (Cell s)
type World r s = (Box r, Tree s, Cell s)

black :: RefM s (Tree s)
black = Leaf <$> R.new Black

gray :: RefM s (Tree s)
gray = Leaf <$> R.new Gray

white :: RefM s (Tree s)
white = Leaf <$> R.new White

-- 0
initial :: Box r -> RefM s (World r s)
{-# SPECIALIZE initial :: Box Double -> RefM s (World Double s) #-}
{-# SPECIALIZE initial :: Box Rational -> RefM s (World Rational s) #-}
initial bounds = (,,) <$> pure bounds <*> gray <*> R.new White

-- 1
refine :: Tree s -> RefM s (Tree s)
refine l@(Leaf r) = do
  c <- R.read r
  case c of
    Gray -> Quad <$> gray <*> gray <*> gray <*> gray
    _ -> pure l
refine (Quad a b c d) =
  Quad <$> refine a <*> refine b <*> refine c <*> refine d

type Graph s = (Map (Cell s) (Set (Cell s)), Map (Cell s) (Set (Cell s)))

empty :: Graph s
```

```
  empty = (M.empty, M.empty)

  insert :: Ref s (Graph s) -> Cell s -> Cell s -> RefM s ()
70 insert g a b = R.modify g $ \(as, bs) ->
    ( M.insertWith S.union a (S.singleton b) as
    , M.insertWith S.union b (S.singleton a) bs
    )

75 intersection :: RealFrac r => World r s -> Box r -> [Cell s]
  {-# SPECIALIZE intersection :: World Double s -> Box Double -> [Cell s] #-}
  {-# SPECIALIZE intersection :: World Rational s -> Box Rational -> [Cell s] #-}
  intersection (bounds, tree, exterior) target
    | not (target `inside` bounds) = [exterior] ++ Q.intersection bounds tree ↙
      ↳ target
80  | otherwise = Q.intersection bounds tree target

  -- 2
  graph :: RealFrac r => (Box r -> Box r) -> World r s -> RefM s (Graph s)
  {-# SPECIALIZE graph :: (Box Double -> Box Double) -> World Double s -> RefM s (↙
    ↳ Graph s) #-}
85 {-# SPECIALIZE graph :: (Box Rational -> Box Rational) -> World Rational s -> ↙
    ↳ RefM s (Graph s) #-}
  graph f world@(bounds, tree, _) = do
    g <- R.new empty
    let go x (Leaf r) = do
          c <- R.read r
90        case c of
            Gray -> forM_ (intersection world (f x)) $ insert g r
            _ -> pure ()
        go x (Quad a b c d) = do
          let (xa, xb, xc, xd) = B.split x
95        go xa a
          go xb b
          go xc c
          go xd d
    go bounds tree
100 R.read g

  -- 3
  whiten :: Graph s -> RefM s ()
  whiten (fwd, rev) = mapM_ visit (M.keys fwd)
105 where
      visit a = do
        c <- R.read a
        when (c == Gray) $ do
          w <- allM (\r -> (White ==) <$> R.read r) (S.toList $ M.findWithDefault ↙
            ↳ S.empty a fwd)
110       when w $ do
            R.write a White
            mapM_ visit (S.toList $ M.findWithDefault S.empty a rev)

  -- 4
115 blacken :: Graph s -> RefM s ()
  blacken (fwd, rev) = mapM_ visit (M.keys fwd) >> mapM_ unmark (M.keys fwd)
    where
      visit a = do
        c <- R.read a
```

```
120         when (c == Gray) $ do
              w <- anyM (\r -> (`elem` [White, Marked]) <$> R.read r) (S.toList $ M.↙
                  ↳ findWithDefault S.empty a fwd)
              when w $ do
                R.write a Marked
                mapM_ visit (S.toList $ M.findWithDefault S.empty a rev)
125       unmark a = R.modify a u
          u White  = White
          u Gray   = Black
          u Black  = Black
          u Marked = Gray
130
      -- 5
      prune :: Tree s -> RefM s (Tree s)
      prune l@(Leaf _) = pure l
      prune (Quad a b c d) = do
135     q <- Quad <$> prune a <*> prune b <*> prune c <*> prune d
        case q of
          Quad (Leaf a) (Leaf b) (Leaf c) (Leaf d) -> do
            a <- R.read a
            b <- R.read b
140         c <- R.read c
            d <- R.read d
            case (a, b, c, d) of
              (Black, Black, Black, Black) -> black
              (White, White, White, White) -> white
145           _ -> pure q
          _ -> pure q

      -- 1-5
      step :: RealFrac r => (Box r -> Box r) -> World r s -> RefM s (World r s)
150   {-# SPECIALIZE step :: (Box Double -> Box Double) -> World Double s -> RefM s (↙
          ↳ World Double s) #-}
      {-# SPECIALIZE step :: (Box Rational -> Box Rational) -> World Rational s -> ↙
          ↳ RefM s (World Rational s) #-}
      step f (bounds, tree, exterior) = do
        tree <- refine tree
        g <- graph f (bounds, tree, exterior)
155     whiten g
        blacken g
        tree <- prune tree
        pure (bounds, tree, exterior)

160   type Image r = Q.Tree (Box r, Color)

      image :: RealFrac r => World r s -> RefM s (Image r)
      {-# SPECIALIZE image :: World Double s -> RefM s (Image Double) #-}
      {-# SPECIALIZE image :: World Rational s -> RefM s (Image Rational) #-}
165   image (bounds, tree, _) = boxed bounds <$> go tree
        where
          go (Leaf l) = Leaf <$> R.read l
          go (Quad a b c d) = Quad <$> go a <*> go b <*> go c <*> go d

170   -- 0-6
      julia :: RealFrac r => (Box r -> Box r) -> r -> [Image r]
      {-# SPECIALIZE julia :: (Box Double -> Box Double) -> Double -> [Image Double] ↙
          ↳ #-}
```

```
{-# SPECIALIZE julia :: (Box Rational -> Box Rational) -> Rational -> [Image ↙
    ↳ Rational] #-}
julia f r = R.run $ do
  world <- initial (B.symmetric r)
  go world
  where
    go world = do
      i <- image world
      world <- step f world
      (i :) <$> go world
```

53 hs/lib/Mandelbrot/Graphics/Trustworthy/Mandelbrot.hs

```
module Mandelbrot.Graphics.Trustworthy.Mandelbrot
  ( Color(..)
  , Image(..)
  , mandelbrot
  ) where


import Control.Monad (forM_)
import Data.Foldable (toList)
import Data.Maybe (mapMaybe)
import qualified Data.Set as S

import Mandelbrot.Graphics.Trustworthy.Julia
import Mandelbrot.Graphics.Trustworthy.Quadratic

import qualified Mandelbrot.Graphics.BoundingBox.D1 as D1
import Mandelbrot.Graphics.BoundingBox.D2
import Mandelbrot.Graphics.QuadTree

mandelbrot :: RealFrac r => Int -> Box r -> [Image r]
{-# SPECIALIZE mandelbrot :: Int -> Box Double -> [Image Double] #-}
{-# SPECIALIZE mandelbrot :: Int -> Box Rational -> [Image Rational] #-}
mandelbrot depth0 bounds = go depth0 (Leaf (bounds, Gray))
  where
    go depth image = image : go (depth + 1) (prune (connected depth <$> refine ↙
        ↳ image))
    er = 2
    origin = symmetric 0
    core t = intersection (symmetric er) (fmap snd t) origin
    connected depth (c, Gray)
      = (,) c
      . color
      . mapMaybe (classify . core)
      . take depth
      . julia (quadratic c)
      $ er
    connected _ p = p

refine :: RealFrac r => Image r -> Image r
{-# SPECIALIZE refine :: Image Double -> Image Double #-}
{-# SPECIALIZE refine :: Image Rational -> Image Rational #-}
refine (Leaf (z, Gray)) =
  let (a, b, c, d) = split z
      l x = Leaf (x, Gray)
```

```
    in  Quad (l a) (l b) (l c) (l d)
refine l@(Leaf _) = l
refine (Quad a b c d) = Quad (refine a) (refine b) (refine c) (refine d)

prune :: Ord r => Image r -> Image r
{-# SPECIALIZE prune :: Image Double -> Image Double #-}
{-# SPECIALIZE prune :: Image Rational -> Image Rational #-}
prune l@(Leaf _) = l
prune (Quad a b c d) = case Quad (prune a) (prune b) (prune c) (prune d) of
    q@(Quad (Leaf (za, a)) (Leaf (zb, b)) (Leaf (zc, c)) (Leaf (zd, d))) ->
      case (a, b, c, d) of
        (Black, Black, Black, Black) -> Leaf (union za zd, Black)
        (White, White, White, White) -> Leaf (union za zd, White)
        _ -> q
    q -> q

classify :: [Color] -> Maybe Color
classify cs
  | S.fromList cs == S.singleton Black = Just Black
  | S.fromList cs == S.singleton White = Just White
  | otherwise = Nothing

color :: [Color] -> Color
color (White:_) = White
color (Black:_) = Black
color _ = Gray
```

54 hs/lib/Mandelbrot/Graphics/Trustworthy/Quadratic.hs

```
module Mandelbrot.Graphics.Trustworthy.Quadratic
  ( quadratic
  ) where

import qualified Mandelbrot.Graphics.BoundingBox.D1 as D1
import Mandelbrot.Graphics.BoundingBox.D2

quadratic :: (Real r) => Box r -> Box r -> Box r
{-# SPECIALIZE quadratic :: Box Double -> Box Double -> Box Double #-}
{-# SPECIALIZE quadratic :: Box Rational -> Box Rational -> Box Rational #-}
quadratic (Box (D1.Box alo ahi) (D1.Box blo bhi)) (Box (D1.Box xlo xhi) (D1.Box ylo yhi)) =
    Box (D1.Box (minimum us) (maximum us)) (D1.Box (minimum vs) (maximum vs))
  where
    (us, vs) = unzip
      [ (x^2-y^2+a, 2*x*y+b)
      | x <- [xlo, xhi]
      , y <- [ylo, yhi]
      , a <- [alo, ahi]
      , b <- [blo, bhi]
      ]
```

55 mandelbrot-graphics.cabal

```
name:                mandelbrot-graphics
version:             0.1.0.0
synopsis:            graphical algorithms related to the Mandelbrot set
```

```
description:         Graphical algorithms related to the Mandelbrot set:
                     image rendering, etc.
homepage:            https://code.mathr.co.uk/mandelbrot-graphics
license:             GPL-3
license-file:        COPYING
author:              Claude Heiland-Allen
maintainer:          claude@mathr.co.uk
copyright:           (c) 2018 Claude Heiland-Allen
category:            Math
build-type:          Simple
cabal-version:       >=1.10
extra-source-files:
  README.md

library
  exposed-modules:
    Mandelbrot.Graphics,
    Mandelbrot.Graphics.Colour,
    Mandelbrot.Graphics.Compute,
    Mandelbrot.Graphics.Matrix,
    Mandelbrot.Graphics.Render,
    Mandelbrot.Graphics.Transform
    Mandelbrot.Graphics.QuadTree
    Mandelbrot.Graphics.BoundingBox.D1
    Mandelbrot.Graphics.BoundingBox.D2
    Mandelbrot.Graphics.STRef.Lazy
    Mandelbrot.Graphics.Trustworthy.Julia
    Mandelbrot.Graphics.Trustworthy.Mandelbrot
    Mandelbrot.Graphics.Trustworthy.Quadratic
  build-depends:
    base >=4.7 && <4.14,
    containers >=0.5 && <0.7,
    mandelbrot-numerics >=0.1 && <0.2,
    monad-loops >=0.4 && <0.5,
    mtl
  hs-source-dirs:      hs/lib
  default-language:    Haskell2010
  ghc-options:         -Wall

executable m-trustworthy-julia
  main-is: m-trustworthy-julia.hs
  build-depends:
    base,
    containers,
    mandelbrot-graphics
  hs-source-dirs: hs/bin
  default-language: Haskell2010
  ghc-options: -Wall -rtsopts

executable m-trustworthy-mandelbrot
  main-is: m-trustworthy-mandelbrot.hs
  build-depends:
    base,
    containers,
    mandelbrot-graphics
  hs-source-dirs: hs/bin
  default-language: Haskell2010
```

```
    ghc-options: -Wall -rtsopts

source-repository head
    type:     git
    location: https://code.mathr.co.uk/mandelbrot-graphics.git

source-repository this
    type:     git
    location: https://code.mathr.co.uk/mandelbrot-graphics.git
    tag:      v0.1.0.0
```

56 README.md

```
mandelbrot-graphics
===================

CPU-based rendering of the Mandelbrot set.
```