mandelbrot-numerics

Claude Heiland-Allen

2015–2019

Contents

1 c/bin/Makefile

```
## mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
## Copyright (C) 2015-2018 Claude Heiland-Allen
## License GPL3+ http://www.gnu.org/licenses/gpl.html

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

PKGCONFIG := PKG_CONFIG_PATH="$(prefix)/lib/pkgconfig" pkg-config
COMPILE := $(CC) -std=c99 -Wall -Wextra -pedantic -fPIC -fopenmp -O3 -pipe -MMD ↙
    ↪ '$(PKGCONFIG) --cflags mandelbrot-numerics mandelbrot-symbolics'
LINK    := $(CC) -fopenmp
LIBS    := '$(PKGCONFIG) --libs mandelbrot-numerics mandelbrot-symbolics sndfile ↙
    ↪ ' -lmpc -lmpfr -lgmp -lm
OBJECTS := $(patsubst %.c,%.o,$(wildcard *.c))
DEPENDS := $(patsubst %.o,%.d,$(OBJECTS))
EXES    := $(patsubst %.o,%,$(OBJECTS))
JSEXES  := $(patsubst %.o,%.js,$(OBJECTS))

all: $(EXES)

js: $(JSEXES)

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

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

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

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

%.js: %.o
        emcc -O2 -o $@ $< ${prefix}/lib/libmandelbrot-numerics.a ${prefix}/lib/ ↙
            ↪ libmpc.a ${prefix}/lib/libmpfr.a ${prefix}/lib/libgmp.a

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

-include $(DEPENDS)
```

2 c/bin/m-attractor.c

```c
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

#include <stdio.h>
#include <mandelbrot-numerics.h>
#include "m-util.h"

static void usage(const char *progname) {
  fprintf
    ( stderr
    , "usage: %s precision z-guess-re z-guess-im c-re c-im period maxsteps\n"
    , progname
    );
}

extern int main(int argc, char **argv) {
  if (argc != 8) {
    usage(argv[0]);
    return 1;
  }
  bool native = true;
  int bits = 0;
  if (! arg_precision(argv[1], &native, &bits)) { return 1; }
  if (native) {
    double zre = 0;
    double zim = 0;
    double cre = 0;
    double cim = 0;
    int period = 0;
    int maxsteps = 0;
    if (! arg_double(argv[2], &zre)) { return 1; }
    if (! arg_double(argv[3], &zim)) { return 1; }
    if (! arg_double(argv[4], &cre)) { return 1; }
    if (! arg_double(argv[5], &cim)) { return 1; }
    if (! arg_int(argv[6], &period)) { return 1; }
    if (! arg_int(argv[7], &maxsteps)) { return 1; }
    complex double z = 0;
    m_newton r = m_d_attractor(&z, zre + I * zim, cre + I * cim, period, ↙
      ↳ maxsteps);
    fprintf(stderr, "result: %s\n", r == m_failed ? "failed" : r == m_stepped ? ↙
      ↳ "stepped" : r == m_converged ? "converged" : "unknown");
    printf("%.16e %.16e\n", creal(z), cimag(z));
    return 0;
  } else {
    mpc_t z_guess, c;
    int period = 0;
    int maxsteps = 0;
    mpc_init2(z_guess, bits);
    mpc_init2(c, bits);
    if (! arg_mpc(argv[2], argv[3], z_guess)) { return 1; }
    if (! arg_mpc(argv[4], argv[5], c)) { return 1; }
    if (! arg_int(argv[6], &period)) { return 1; }
    if (! arg_int(argv[7], &maxsteps)) { return 1; }
    mpc_t z_out;
    mpc_init2(z_out, bits);
    m_newton r = m_r_attractor(z_out, z_guess, c, period, maxsteps);
```

```
    fprintf(stderr, "result: %s\n", r == m_failed ? "failed" : r == m_stepped ? ↙
        ↳ "stepped" : r == m_converged ? "converged" : "unknown");
    mpfr_printf("%Re %Re\n", mpc_realref(z_out), mpc_imagref(z_out));
    mpc_clear(z_out);
    mpc_clear(z_guess);
    mpc_clear(c);
    return 0;
  }
  return 1;
}
```

3 c/bin/m-ball-period.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

#include <stdio.h>
#include <mandelbrot-numerics.h>
#include "m-util.h"

static void usage(const char *progname) {
  fprintf
    ( stderr
    , "usage:\n%s precision center-re center-im radius maxperiod\n"
      "%s precision period radius maxperiod <<EOF\n"
      "center-re\ncenter-im\nEOF\n"
    , progname, progname
    );
}

extern int main(int argc, char **argv) {
  if (argc != 4 && argc != 6) {
    usage(argv[0]);
    return 1;
  }
  bool native = true;
  int bits = 0;
  if (! arg_precision(argv[1], &native, &bits)) { return 1; }
  if (native) {
    double cre = 0;
    double cim = 0;
    double radius = 0;
    int maxperiod = 0;
    if (argc == 4)
    {
      if (! stdin_double(&cre)) { return 1; }
      if (! stdin_double(&cim)) { return 1; }
      if (! arg_double(argv[2], &radius)) { return 1; }
      if (! arg_int(argv[3], &maxperiod)) { return 1; }
    }
    else
    {
      if (! arg_double(argv[2], &cre)) { return 1; }
      if (! arg_double(argv[3], &cim)) { return 1; }
      if (! arg_double(argv[4], &radius)) { return 1; }
      if (! arg_int(argv[5], &maxperiod)) { return 1; }
```

```
45      }
        int period = m_d_ball_period_do(cre + I * cim, radius, maxperiod);
        if (period > 0) {
          printf("%d\n", period);
          return 0;
50      }
      } else {
        mpc_t center;
        mpfr_t radius;
        int maxperiod = 0;
55      mpc_init2(center, bits);
        mpfr_init2(radius, 53);
        if (argc == 4)
        {
          if (! stdin_mpfr(mpc_realref(center))) { return 1; }
60        if (! stdin_mpfr(mpc_imagref(center))) { return 1; }
          if (! arg_mpfr(argv[2], radius)) { return 1; }
          if (! arg_int(argv[3], &maxperiod)) { return 1; }
        }
        else
65      {
          if (! arg_mpc(argv[2], argv[3], center)) { return 1; }
          if (! arg_mpfr(argv[4], radius)) { return 1; }
          if (! arg_int(argv[5], &maxperiod)) { return 1; }
        }
70      int period = m_r_ball_period_do(center, radius, maxperiod);
        if (period > 0) {
          printf("%d\n", period);
        }
        mpc_clear(center);
75      mpfr_clear(radius);
        return period <= 0;
      }
      return 1;
    }
```

4 c/bin/m-ball-vs-box-period.c

```
    #include <complex.h>
    #include <math.h>
    #include <stdio.h>
    #include <stdlib.h>
5
    #define N 36
    #define PI 3.141592653589793

    static int sgn(double x)
10  {
      return (x > 0) - (0 > x);
    }

    static double cross(double _Complex a, double _Complex b) {
15    return cimag(a) * creal(b) - creal(a) * cimag(b);
    }

    static int crosses_positive_real_axis(double _Complex a, double _Complex b) {
      if (sgn(cimag(a)) != sgn(cimag(b))) {
```

```
      double _Complex d = b - a;
      int s = sgn(cimag(d));
      int t = sgn(cross(d, a));
      return s == t;
    }
  return 0;
}

int main(int argc, char **argv)
{
  if (argc < 4) return 1;
  double _Complex c = atof(argv[1]) + I * atof(argv[2]);
  double r = atof(argv[3]);
  double _Complex c_ball = c;
  double _Complex z_ball = c;
  double r_ball = r;
  double _Complex c_box[N];
  double _Complex z_box[N];
  for (int i = 0; i < N; ++i)
  {
    double t = 2 * PI * (i + 0.5) / N;
    c_box[i] = z_box[i] = c + r * (cos(t) + I * sin(t));
  }
  int ball_found = 0;
  int box_found = 0;
  int both_found = 0;
  for (int p = 1; p < 100; ++p)
  {
    double a_box = 0;
    for (int i = 0; i < N; ++i)
    {
      a_box += creal(z_box[i]) * cimag(z_box[(i+1)%N]) - creal(z_box[(i+1)%N]) *↙
          ↳ cimag(z_box[i]);
    }
    a_box = 0.5 * fabs(a_box);
    double r_box = sqrt(a_box / PI);
    int p_ball = cabs(z_ball) <= r_ball;
    int count = 0;
    for (int i = 0; i < N; ++i)
    {
      count += crosses_positive_real_axis(z_box[i], z_box[(i+1)%N]);
    }
    int p_box = count & 1;
    ball_found |= p_ball;
    box_found |= p_box;
    both_found = ball_found && box_found;
    printf("%d\t%e\t%e\t%d\t%d\n", p, r_ball, r_box, p_ball, p_box);
    fflush(stdout);
    if (both_found) break;
    r_ball = (2 * cabs(z_ball) + r_ball) * r_ball;
    z_ball = z_ball * z_ball + c_ball;
    for (int i = 0; i < N; ++i)
    {
      z_box[i] = z_box[i] * z_box[i] + c_box[i];
    }
  }
  return 0;
```

```
}
```

5 c/bin/m-box-misiurewicz.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

#include <stdio.h>
#include <mandelbrot-numerics.h>
#include "m-util.h"

static void usage(const char *progname) {
  fprintf
    ( stderr
    , "usage: %s precision center-re center-im radius maxpreperiod maxperiod\n"
    , progname
    );
}

extern int main(int argc, char **argv) {
  if (argc != 7) {
    usage(argv[0]);
    return 1;
  }
  bool native = true;
  int bits = 0;
  if (! arg_precision(argv[1], &native, &bits)) { return 1; }
  if (native) {
    double cre = 0;
    double cim = 0;
    double radius = 0;
    int maxpreperiod = 0;
    int maxperiod = 0;
    if (! arg_double(argv[2], &cre)) { return 1; }
    if (! arg_double(argv[3], &cim)) { return 1; }
    if (! arg_double(argv[4], &radius)) { return 1; }
    if (! arg_int(argv[5], &maxpreperiod)) { return 1; }
    if (! arg_int(argv[6], &maxperiod)) { return 1; }
    int preperiod = -1;
    int period = 0;
    bool ok = m_d_box_misiurewicz_do(&preperiod, &period, cre + I * cim, radius,↙
        ↳ maxpreperiod, maxperiod);
    if (ok) {
      printf("%d %d\n", preperiod, period);
    }
    return ok ? 0 : 1;
  } else {
    mpc_t center;
    mpfr_t radius;
    int maxpreperiod = 0;
    int maxperiod = 0;
    mpc_init2(center, bits);
    mpfr_init2(radius, bits);
    if (! arg_mpc(argv[2], argv[3], center)) { return 1; }
    if (! arg_mpfr(argv[4], radius)) { return 1; }
    if (! arg_int(argv[5], &maxpreperiod)) { return 1; }
```

```
    if (! arg_int(argv[6], &maxperiod)) { return 1; }
    int preperiod = -1;
    int period = 0;
    bool ok = m_r_box_misiurewicz_do(&preperiod, &period, center, radius, ↙
      ↳ maxpreperiod, maxperiod);
    if (ok) {
      printf("%d %d\n", preperiod, period);
    }
    mpc_clear(center);
    mpfr_clear(radius);
    return ok ? 0 : 1;
  }
  return 1;
}
```

6 c/bin/m-box-period.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

#include <stdio.h>
#include <mandelbrot-numerics.h>
#include "m-util.h"

static void usage(const char *progname) {
  fprintf
    ( stderr
    , "usage: %s precision center-re center-im radius maxperiod\n"
    , progname
    );
}

extern int main(int argc, char **argv) {
  if (argc != 6) {
    usage(argv[0]);
    return 1;
  }
  bool native = true;
  int bits = 0;
  if (! arg_precision(argv[1], &native, &bits)) { return 1; }
  if (native) {
    double cre = 0;
    double cim = 0;
    double radius = 0;
    int maxperiod = 0;
    if (! arg_double(argv[2], &cre)) { return 1; }
    if (! arg_double(argv[3], &cim)) { return 1; }
    if (! arg_double(argv[4], &radius)) { return 1; }
    if (! arg_int(argv[5], &maxperiod)) { return 1; }
    int period = m_d_box_period_do(cre + I * cim, radius, maxperiod);
    if (period > 0) {
      printf("%d\n", period);
      return 0;
    }
  } else {
    mpc_t center;
```

```
    mpfr_t radius;
    int maxperiod = 0;
    mpc_init2(center, bits);
    mpfr_init2(radius, bits);
    if (! arg_mpc(argv[2], argv[3], center)) { return 1; }
    if (! arg_mpfr(argv[4], radius)) { return 1; }
    if (! arg_int(argv[5], &maxperiod)) { return 1; }
    int period = m_r_box_period_do(center, radius, maxperiod);
    if (period > 0) {
      printf("%d\n", period);
    }
    mpc_clear(center);
    mpfr_clear(radius);
    return period <= 0;
  }
  return 1;
}
```

7 c/bin/m-circles.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

// m-circles.c (C) 2016 Claude Heiland-Allen <claude@mathr.co.uk>
// demonstrate distortion of some circle-like components in the Mandelbrot set
// see: http://math.stackexchange.com/q/1857237/209286

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

#include <mandelbrot-numerics.h>
// https://code.mathr.co.uk/mandelbrot-numerics
//
// m_d_interior(z_out, c_out, z_guess, c_guess, coordinate, period, steps);
// Find a point with given interior 'coordinate' in a hyperbolic component of
// 'period' (magnitude 1 is boundary).
// Uses Newton's method with at most 'steps'.
//
// m_d_nucleus(c_out, c_guess, period, steps);
// Find the nucleus of a hyperbolic component with given 'period'.
// Uses Newton's method with at most 'steps'.
//
// m_d_size(nucleus, period);
// Find a size estimate for a hyperbolic component.
// For circular-like components, approximates the diameter.

#define twopi 6.283185307179586

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

  // initial component
  int period = 2;
  double size = 0.5;
  complex double nucleus = -1;
```

```
  complex double leftpoint = -1.25;
  complex double rightpoint = -0.75;

  // consider components heading towards -inf on the period-doubling cascade
  for (int depth = 0; depth < 12; ++depth) {

    // geometric midpoint might differ from nucleus in general
    complex double midpoint = (leftpoint + rightpoint) / 2;
    double radius = cabs(rightpoint - midpoint);

    // trace boundaries of components
    complex double z = nucleus, c = nucleus;
    for (int t = 0; t < 360; ++t) {

      // m_d_interior is unstable at root point (theta = 0), hence adding 0.5deg
      double theta = twopi * (t + 0.5) / 360;
      m_d_interior(&z, &c, z, c, cexp(I * theta), period, 64);

      // compute the geometric distance to the midpoint
      // divide by the geometric radius to normalize output range
      double distance = cabs(c - midpoint) / radius;

      // output with full precision
      printf("%.18g\n", distance);
    }

    // separator between curves for gnuplot
    printf("\n\n");

    // advance to the next component
    period *= 2;

    // the next component is approximately 1/4 the size
    complex double estimate = nucleus - 1.25 * (size / 2);
    m_d_nucleus(&nucleus, estimate, period, 64);
    size = cabs(m_d_size(nucleus, period));

    // update edges - can't find rightpoint with m_d_interior due to instability
    rightpoint = leftpoint;
    m_d_interior(&z, &leftpoint, nucleus, nucleus, -1, period, 64);

  }

  return 0;
}
```

8 c/bin/m-describe.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2019 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

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

```
#include <string.h>
#include <mandelbrot-numerics.h>
#include "m-util.h"

static const char *compass[16] =
  { "east"
  , "east-north-east"
  , "north-east"
  , "north-north-east"
  , "north"
  , "north-north-west"
  , "north-west"
  , "west-north-west"
  , "west"
  , "west-south-west"
  , "south-west"
  , "south-south-west"
  , "south"
  , "south-south-east"
  , "south-east"
  , "east-south-east"
  };

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

int main(int argc, char **argv)
{
  if (argc < 7)
  {
    fprintf(stderr, "usage: %s precision maxperiod maxiters re im ncpus\n", argv[0]);
    return 1;
  }
  bool native = true;
  int bits = 0;
  int maxperiod = 0;
  int maxiters = 0;
  int ncpus = 0;
  if (! arg_precision(argv[1], &native, &bits)) { return 1; }
  if (! arg_int(argv[2], &maxperiod)) { return 1; }
  if (! arg_int(argv[3], &maxiters)) { return 1; }
  if (native) {
    double cre = 0;
    double cim = 0;
    if (! arg_double(argv[4], &cre)) { return 1; }
    if (! arg_double(argv[5], &cim)) { return 1; }
    if (! arg_int(argv[6], &ncpus)) { return 1; }

    double _Complex c = cre + I * cim;
    double _Complex z = 0;
    double _Complex dc = 0;
    printf("the input point was %+.18f + %+.18f i\n", creal(c), cimag(c));
    bool escaped = false;
    for (int i = 0; i < maxiters; ++i)
    {
```

```
      dc = 2 * z * dc + 1;
      z = z * z + c;
      if (!escaped && cabs2(z) > 1e10)
      {
        double mu = i + 1 - log2(log(cabs(z)));
        double de = 2 * cabs(z) * log(cabs(z)) / cabs(dc);
        printf("the point escaped with dwell %.5f\nand exterior distance ↙
            ↳ estimate %.5g\n", mu, de);
        escaped = true;
      }
    }
    if (! escaped)
    {
      printf("the point didn't escape after %d iterations\n", maxiters);
    }
    printf("\nnearby hyperbolic components to the input point:\n");
    double _Complex cperiodic = 0;
    double periodic = 1.0 / 0.0;
    int lastlastperiod = 1;
    int lastperiod = 1;
    bool interior = false;
    z = 0;
    for (int period = 1; period <= maxperiod && !interior; ++period)
    {
      z = z * z + c;
      double m = cabs(z);
      if (m <= periodic)
      {
        periodic = m;
        double _Complex q;
        if (m_failed != m_d_nucleus(&q, c, period, 64))
        {
          double _Complex w = 0;
          for (int i = 0; i <= 64; ++i)
          {
            m_d_attractor(&w, w, (cperiodic * (64 - i) + i * q) / 64, lastperiod↙
                ↳ , 64);
          }
          double _Complex dw = 1;
          for (int i = 0; i < lastperiod; ++i)
          {
            dw *= 2 * w;
            w = w * w + q;
          }
          double _Complex nucleus_domain_coordinate = dw;
          double _Complex u = 0, v = 0;
          for (int i = 1; i <= lastperiod; ++i)
          {
            u = u * u + q;
            if (i == lastlastperiod)
              v = u;
          }
          double _Complex atom_domain_coordinate = u / v;
          cperiodic = q;
          lastlastperiod = lastperiod;
          lastperiod = period;
          int dir = round(carg(q - c) / 6.283185307179586 * 16);
```

```
  dir += 16;
  dir %= 16;
  double _Complex r = m_d_size(q, period);
  int sdir = round(carg(r) / 6.283185307179586 * 16);
  sdir += 16;
  sdir += 8;
  sdir %= 16;
  int cardioid = m_d_shape_discriminant(m_d_shape_estimate(q, period)) ↙
      ↳ == m_cardioid;
  printf(
    "\n- a period %d %s\n"
    "  with nucleus at %+.18f + %+.18f i\n"
    "  the component has size %.5g and is pointing %s\n"
    "  the atom domain has size %.5g\n"
    "  the nucleus domain coordinate is %.5g at turn %.18f\n"
    "  the atom    domain coordinate is %.5g at turn %.18f\n"
    "  the nucleus is %.5g to the %s of the input point\n"
    , period, cardioid ? "cardioid" : "circle"
    , creal(q), cimag(q)
    , cabs(r), compass[sdir]
    , m_d_domain_size(q, period)
    , cabs(nucleus_domain_coordinate), fmod(carg(↙
        ↳ nucleus_domain_coordinate) / 6.283185307179586 + 1, 1)
    , cabs(atom_domain_coordinate), fmod(carg(atom_domain_coordinate) / ↙
        ↳ 6.283185307179586 + 1, 1)
    , cabs(q - c), compass[dir]
    );
  if (m_failed != m_d_attractor(&w, z, c, period, 64))
  {
    double _Complex w0 = w;
    double _Complex dw = 1;
    for (int i = 0; i < period; ++i)
    {
      dw = 2 * w * dw;
      w = w * w + c;
    }
    double t = carg(dw) / 6.283185307179586;
    t -= floor(t);
    interior = cabs(dw) <= 1;
    printf(
      "  the input point is %s to this component at\n"
      "  radius %.5g and angle %.5f (in turns)\n"
      "  a point in the attractor is %+.18f + %+.18f i\n"
      , interior ? "interior" : "exterior"
      , cabs(dw), t
      , creal(w0), cimag(w0)
      );
    if (interior)
    {
      double ide = 0;
      double _Complex dz = 0;
      if (m_d_interior_de(&ide, &dz, w0, c, period, 64))
      {
        printf(
          "  the interior distance estimate is %.5g\n"
          , ide
          );
```

```
175                 }
                  }
                }
                char *lo = 0, *hi = 0;
                m_d_external_angles(&lo, &hi, q, period, 0.9 - 0.8 / sqrt(period), 64)↙
                    ↳ ;
180             if (lo && hi)
                  printf("  external angles of this component are:\n  %s\n  %s\n", lo,↙
                      ↳  hi);
                if (lo)
                  free(lo);
                if (hi)
185               free(hi);
              }
            }
          }

190       printf("\nnearby Misiurewicz points to the input point:\n");
          double de = 1.0 / 0.0;
          double d = 1.0 / 0.0;
          int period = -1;
          int preperiod = -1;
195       for (int m = 1; m <= lastperiod; ++m)
          {
            z = c;
            double _Complex w = c;
            for (int n = 0; n < m; ++n)
200         {
              z = z * z + c;
            }
            for (int n = 1; n <= lastperiod; ++n)
            {
205           z = z * z + c;
              w = w * w + c;
              double z2 = cabs2(z - w);
              if (z2 < de)
              {
210             de = z2;
                period = m;
                preperiod = n + 1;

                double _Complex z2 = c;
215             m_d_misiurewicz(&z2, z2, preperiod, period, 64);
                m_d_misiurewicz_naive(&z2, z2, preperiod, period, 64);
                double d2 = cabs2(z2 - c);
                if (d2 < d)
                {
220               d = d2;
                  int dir = round(carg(z2 - c) / 6.283185307179586 * 16);
                  dir += 16;
                  dir %= 16;
                  double _Complex w = z2;
225               double _Complex dw = 1;
                  for (int i = 0; i < preperiod; ++i)
                  {
                    w = w * w + z2;
                  }
```

```
230                 for (int i = 0; i < period; ++i)
                    {
                      dw = 2 * w * dw;
                      w = w * w + z2;
                    }
235                 double t = carg(dw) / 6.283185307179586;
                    if (cabs(dw) > 1)
                    {
                      printf(
                        "\n- %dp%d\n"
240                     "  the strength is %.5g\n",
                        preperiod, period,
                        de
                        );
                      printf(
245                     "  the center is at %+.18f + %.18f i\n"
                        "  the center is %.5g to the %s of the input point\n"
                        "  the multiplier has radius %.5g and angle %.5f (in turns)\n",
                        creal(z2), cimag(z2),
                        cabs(z2 - c), compass[dir],
250                     cabs(dw), t
                        );
                    }
                  }
                }
255           }
        }

      } else {
        mpc_t c;
260     mpc_init2(c, bits);
        if (! arg_mpc(argv[4], argv[5], c)) { return 1; }
        if (! arg_int(argv[6], &ncpus)) { return 1; }

        mpc_t z, dc, q, r, w, w0, dw, periodicz, a;
265     mpfr_t z2, er2, de, periodic;
        mpc_init2(z, bits);
        mpc_init2(dc, bits);
        mpc_init2(q, bits);
        mpc_init2(r, bits);
270     mpc_init2(w, bits);
        mpc_init2(w0, bits);
        mpc_init2(dw, bits);
        mpc_init2(periodicz, bits);
        mpc_init2(a, 53);
275     mpfr_init2(z2, 53);
        mpfr_init2(er2, 53);
        mpfr_init2(de, 53);
        mpfr_init2(periodic, 53);
        mpfr_set_d(er2, 1e100, MPFR_RNDN);
280     mpc_set_ui_ui(z, 0, 0, MPC_RNDNN);
        mpc_set_ui_ui(dc, 0, 0, MPC_RNDNN);
        mpfr_printf("the input point was %+Re + %+Re i\n", mpc_realref(c), ↙
            ↳ mpc_imagref(c));
        bool escaped = false;
        for (int i = 0; i < maxiters; ++i)
285     {
```

```
      // dc = 2 * z * dc + 1;
      mpc_mul(dc, dc, z, MPC_RNDNN);
      mpfr_mul_2exp(mpc_realref(dc), mpc_realref(dc), 1, MPFR_RNDN);
      mpfr_mul_2exp(mpc_imagref(dc), mpc_imagref(dc), 1, MPFR_RNDN);
290   mpfr_add_ui(mpc_realref(dc), mpc_realref(dc), 1, MPFR_RNDN);
      // z = z * z + c;
      mpc_sqr(z, z, MPC_RNDNN);
      mpc_add(z, z, c, MPC_RNDNN);
      mpc_norm(z2, z, MPFR_RNDN);
295   if (mpfr_greater_p(z2, er2))
      {
        double dz2 = mpfr_get_d(z2, MPFR_RNDN);
        double mu = i + 1 - log2(log(sqrt(dz2)));
        mpfr_set_d(de, 2 * sqrt(dz2) * log(sqrt(dz2)), MPFR_RNDN);
300     mpc_abs(z2, dc, MPFR_RNDN);
        mpfr_div(de, de, z2, MPFR_RNDN);
        mpfr_printf("the point escaped with dwell %.5f\nand exterior distance ↙
            ↳ estimate %.5Re\n", mu, de);
        escaped = true;
        break;
305   }
    }
    if (! escaped)
    {
      printf("the point didn't escape after %d iterations\n", maxiters);
310 }
    printf("nearby hyperbolic components to the input point:\n");
    mpfr_set_d(periodic, 1.0 / 0.0, MPFR_RNDN);

    int lastperiod = 1;
315 bool interior = false;
    mpc_set_ui_ui(z, 0, 0, MPC_RNDNN);
    for (int period = 1; period <= maxperiod && !interior; ++period)
    {
      // z = z * z + c;
320   mpc_sqr(z, z, MPC_RNDNN);
      mpc_add(z, z, c, MPC_RNDNN);
      mpc_norm(z2, z, MPFR_RNDN);
      if (! mpfr_greater_p(z2, periodic))
      {
325     mpc_div(a, z, periodicz, MPC_RNDNN);
        double _Complex atom = mpc_get_dc(a, MPC_RNDNN);
        int adir = round(carg(atom) / 6.283185307179586 * 16);
        adir += 16;
        adir %= 16;
330     mpc_set(periodicz, z, MPC_RNDNN);
        mpfr_set(periodic, z2, MPFR_RNDN);
        if (m_failed != m_r_nucleus(q, c, period, 64, ncpus))
        {
          mpc_sub(dc, q, c, MPC_RNDNN);
335       mpc_arg(z2, dc, MPFR_RNDN);
          mpc_abs(er2, dc, MPFR_RNDN);
          int dir = round(mpfr_get_d(z2, MPFR_RNDN) / 6.283185307179586 * 16);
          dir += 16;
          dir %= 16;
340       m_r_size(r, q, period);
          mpc_arg(z2, r, MPFR_RNDN);
```

```
        int sdir = round(mpfr_get_d(z2, MPFR_RNDN) / 6.283185307179586 * 16);
        sdir += 16;
        sdir += 8;
345     sdir %= 16;
        mpc_abs(z2, r, MPFR_RNDN);
        mpfr_exp_t e = mpfr_get_exp(z2);
        mpfr_prec_t prec = 16 - e;
        mpfr_prec_round(mpc_realref(q), prec, MPFR_RNDN);
350     mpfr_prec_round(mpc_imagref(q), prec, MPFR_RNDN);
        m_r_domain_size(de, q, period);
        mpfr_printf(
          "\n- a period %d %s\n"
          "  with nucleus at %+Re + %+Re i\n"
355       "  the component has size %.5Re and is pointing %s\n"
          "  the atom domain has size %.5Re\n"
          "  the atom domain coordinates of the input point are %+.5g + %+.5g ↙
            ↳ i\n"
          "  the atom domain coordinates in polar form are %.5g to the %s\n"
          "  the nucleus is %.5Re to the %s of the input point\n"
360       , period, m_r_shape(q, period) == m_cardioid ? "cardioid" : "circle"
          , mpc_realref(q), mpc_imagref(q)
          , z2, compass[sdir]
          , de
          , creal(atom), cimag(atom)
365       , cabs(atom), compass[adir]
          , er2, compass[dir]
          );
        lastperiod = period;
        if (! interior)
370     {
          if (m_failed != m_r_attractor(w, z, c, period, 64))
          {
            mpc_set_prec(w0, prec);
            mpc_set(w0, w, MPC_RNDNN);
375         mpc_set_ui_ui(dw, 1, 0, MPC_RNDNN);
            for (int i = 0; i < period; ++i)
            {
              // dw = 2 * w * dw;
              mpc_mul(dw, w, dw, MPC_RNDNN);
380           mpfr_mul_2exp(mpc_realref(dw), mpc_realref(dw), 1, MPFR_RNDN);
              mpfr_mul_2exp(mpc_imagref(dw), mpc_imagref(dw), 1, MPFR_RNDN);
              // w = w * w + c;
              mpc_sqr(w, w, MPC_RNDNN);
              mpc_add(w, w, c, MPC_RNDNN);
385         }
            mpc_arg(z2, dw, MPFR_RNDN);
            double t = mpfr_get_d(z2, MPFR_RNDN) / 6.283185307179586;
            t -= floor(t);
            mpc_abs(z2, dw, MPFR_RNDN);
390         interior = mpfr_get_d(z2, MPFR_RNDN) <= 1;
            mpfr_printf(
              "  the input point is %s to this component at\n"
              "  radius %.5Re and angle %.18f (in turns)\n"
              "  the multiplier is %+.5Re + %+.5Re i\n"
395           "  a point in the attractor is %+Re + %+Re i\n"
              , interior ? "interior" : "exterior"
              , z2, t
```

```
                , mpc_realref(dw), mpc_imagref(dw)
                , mpc_realref(w0), mpc_imagref(w0)
400             );
            }
          }
          fflush(stdout);

405       if (period < 200)
          {
            char *lo = 0, *hi = 0;
            m_r_external_angles(&lo, &hi, q, period, 0.5 - 0.4 / sqrt(period), ↙
                ↳ 64);
            if (lo && hi)
410           printf("  external angles of this component are:\n  %s\n  %s\n", ↙
                  ↳ lo, hi);
            if (lo)
              free(lo);
            if (hi)
              free(hi);
415         fflush(stdout);
          }

        }
      }
420   }
      (void) lastperiod;

#if 0
      if (! interior)
425   {
        mpfr_set_d(de, 1.0 / 0.0, MPFR_RNDN);
        int period = -1;
        int preperiod = -1;
        for (int m = 1; m < maxperiod; ++m)
430     {
          if (lastperiod > 1 && (m % lastperiod) == 0)
          {
            continue;
          }
435       mpc_set(z, c, MPC_RNDNN);
          mpc_set(w, c, MPC_RNDNN);
          for (int n = 0; n < m; ++n)
          {
            // z = z * z + c;
440         mpc_sqr(z, z, MPC_RNDNN);
            mpc_add(z, z, c, MPC_RNDNN);
          }
          for (int n = 0; n < maxperiod; ++n)
          {
445         mpc_sub(dw, z, w, MPC_RNDNN);
            mpc_norm(z2, dw, MPFR_RNDN);
            if (mpfr_less_p(z2, de))
            {
              mpfr_set(de, z2, MPFR_RNDN);
450           period = m;
              preperiod = n;
            }
```

```
      // z = z * z + c;
      mpc_sqr(z, z, MPC_RNDNN);
      mpc_add(z, z, c, MPC_RNDNN);
      // w = w * w + c;
      mpc_sqr(w, w, MPC_RNDNN);
      mpc_add(w, w, c, MPC_RNDNN);
    }
  }

  printf("\nnearby Misiurewicz points to the input point:\n");
  mpfr_printf(
    "\n- %dp%d\n"
    "  the strength is %.5Re\n",
    preperiod, period,
    de
    );
  mpc_set(z, c, MPC_RNDNN);
  m_r_misiurewicz(z, z, preperiod, period, 64);
  m_r_misiurewicz_naive(z, z, preperiod, period, 64);
  m_r_misiurewicz(z, z, preperiod, period, 64);
  m_r_misiurewicz_naive(z, z, preperiod, period, 64);
  {
    mpc_sub(w, z, c, MPC_RNDNN);
    mpc_arg(de, w, MPFR_RNDN);
    int dir = round(mpfr_get_d(de, MPFR_RNDN) / 6.283185307179586 * 16);
    dir += 16;
    dir %= 16;
    mpc_set(w, z, MPC_RNDNN);
    mpc_set_ui_ui(dw, 1, 0, MPC_RNDNN);
    for (int i = 0; i < preperiod; ++i)
    {
      // w = w * w + z;
      mpc_sqr(w, w, MPC_RNDNN);
      mpc_add(w, w, z, MPC_RNDNN);
    }
    for (int i = 0; i < period; ++i)
    {
      // dw = 2 * w * dw;
      mpc_mul(dw, w, dw, MPC_RNDNN);
      mpfr_mul_2exp(mpc_realref(dw), mpc_realref(dw), 1, MPFR_RNDN);
      mpfr_mul_2exp(mpc_imagref(dw), mpc_imagref(dw), 1, MPFR_RNDN);
      // w = w * w + z;
      mpc_sqr(w, w, MPC_RNDNN);
      mpc_add(w, w, z, MPC_RNDNN);
    }
    mpc_arg(de, w, MPFR_RNDN);
    double t = mpfr_get_d(de, MPFR_RNDN) / 6.283185307179586;
    mpc_abs(de, w, MPFR_RNDN);
    mpc_sub(w, z, c, MPC_RNDNN);
    mpc_abs(z2, w, MPFR_RNDN);
    mpfr_printf(
      "  the center is at %+Re + %+Re i\n"
      "  the center is %.5Re to the %s of the input point\n"
      "  the multiplier has radius %.18Rg and angle %.18f (in turns)\n",
      mpc_realref(z), mpc_imagref(z),
      z2, compass[dir],
      de, t
```

```
            );
        }
        fflush(stdout);
      }
#endif

      mpc_clear(c);
      mpc_clear(z);
      mpc_clear(dc);
      mpc_clear(q);
      mpc_clear(r);
      mpc_clear(w);
      mpc_clear(w0);
      mpc_clear(dw);
      mpfr_clear(z2);
      mpfr_clear(er2);
      mpfr_clear(de);
      mpfr_clear(periodic);
    }

  return 0;
}
```

9 c/bin/m-domain-coord.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

#include <stdio.h>
#include <mandelbrot-numerics.h>
#include "m-util.h"

static void usage(const char *progname) {
  fprintf
    ( stderr
    , "usage: %s precision guess-re guess-im coord-re coord-im loperiod hiperiod
      ↪ maxsteps\n"
    , progname
    );
}

extern int main(int argc, char **argv) {
  if (argc != 9) {
    usage(argv[0]);
    return 1;
  }
  bool native = true;
  int bits = 0;
  if (! arg_precision(argv[1], &native, &bits)) { return 1; }
  if (native) {
    double cre = 0;
    double cim = 0;
    double are = 0;
    double aim = 0;
    int loperiod = 0;
    int hiperiod = 0;
```

```
    int maxsteps = 0;
    if (! arg_double(argv[2], &cre)) { return 1; }
    if (! arg_double(argv[3], &cim)) { return 1; }
    if (! arg_double(argv[4], &are)) { return 1; }
    if (! arg_double(argv[5], &aim)) { return 1; }
    if (! arg_int(argv[6], &loperiod)) { return 1; }
    if (! arg_int(argv[7], &hiperiod)) { return 1; }
    if (! arg_int(argv[8], &maxsteps)) { return 1; }
    double _Complex c = 0;
    m_d_domain_coord(&c, cre + I * cim, are + I * aim, loperiod, hiperiod, ↙
      ↳ maxsteps);
    printf("%.16e %.16e\n", creal(c), cimag(c));
    return 0;
  } else {
    mpc_t c_guess;
    mpc_t domain_coord;
    int loperiod = 0;
    int hiperiod = 0;
    int maxsteps = 0;
    mpc_init2(c_guess, bits);
    mpc_init2(domain_coord, bits);
    if (! arg_mpc(argv[2], argv[3], c_guess)) { return 1; }
    if (! arg_mpc(argv[4], argv[5], domain_coord)) { return 1; }
    if (! arg_int(argv[6], &loperiod)) { return 1; }
    if (! arg_int(argv[7], &hiperiod)) { return 1; }
    if (! arg_int(argv[8], &maxsteps)) { return 1; }
    mpc_t c_out;
    mpc_init2(c_out, bits);
    m_r_domain_coord(c_out, c_guess, domain_coord, loperiod, hiperiod, maxsteps)↙
      ↳ ;
    mpfr_printf("%Re %Re\n", mpc_realref(c_out), mpc_imagref(c_out));
    mpc_clear(c_out);
    mpc_clear(domain_coord);
    mpc_clear(c_guess);
    return 0;
  }
  return 1;
}
```

10 c/bin/m-domain-size.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

#include <stdio.h>
#include <mandelbrot-numerics.h>
#include "m-util.h"

static void usage(const char *progname) {
  fprintf
    ( stderr
    , "usage: %s precision nucleus-re nucleus-im period\n"
    , progname
    );
}
```

```
extern int main(int argc, char **argv) {
  if (argc != 5) {
    usage(argv[0]);
    return 1;
  }
  bool native = true;
  int bits = 0;
  if (! arg_precision(argv[1], &native, &bits)) { return 1; }
  if (native) {
    double nre = 0;
    double nim = 0;
    int period = 0;
    if (! arg_double(argv[2], &nre)) { return 1; }
    if (! arg_double(argv[3], &nim)) { return 1; }
    if (! arg_int(argv[4], &period)) { return 1; }
    double size = m_d_domain_size(nre + I * nim, period);
    printf("%.16e\n", size);
    return 0;
  } else {
    mpc_t n;
    int period = 0;
    mpc_init2(n, bits);
    if (! arg_mpc(argv[2], argv[3], n)) { return 1; }
    if (! arg_int(argv[4], &period)) { return 1; }
    mpfr_t size;
    mpfr_init2(size, bits);
    m_r_domain_size(size, n, period);
    mpfr_printf("%Re\n", size);
    mpfr_clear(size);
    mpc_clear(n);
    return 0;
  }
  return 1;
}
```

11 c/bin/m-ejs-evotree-morph.c

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

#define NCPUS 8

void m_r_domain_coord_of_point(mpc_t out, const mpc_t c, int lo, int hi)
{
  mpfr_prec_t precr = mpfr_get_prec(mpc_realref(c));
  mpfr_prec_t preci = mpfr_get_prec(mpc_imagref(c));
  mpfr_prec_t prec = precr > preci ? precr : preci;
  mpc_t z, w;
  mpc_init2(z, prec);
  mpc_init2(w, prec);
  mpc_set_si(z, 0, MPC_RNDNN);
  int i;
  for (i = 0; i < lo; ++i)
  {
    mpc_sqr(z, z, MPC_RNDNN);
    mpc_add(z, z, c, MPC_RNDNN);
  }
```

```
  mpc_set(w, z, MPC_RNDNN);
  for (; i < hi; ++i)
  {
    mpc_sqr(w, w, MPC_RNDNN);
    mpc_add(w, w, c, MPC_RNDNN);
  }
  mpc_div(out, w, z, MPC_RNDNN);
  mpc_clear(w);
  mpc_clear(z);
}

int main()
{
  // increase COUNT += 2 makes runtime *= 3
#define COUNT 23
  mpc_t n[COUNT];
  int p[COUNT];
  int k = 0;
  int newton_steps = 64;
#define EXAMPLE_2
#ifdef EXAMPLE_1
  int arm_length = 3;
  int misiurewicz_offset = -38;
  p[k] =    3; mpc_init2(n[k],  58); mpc_set_str(n[k], "(-1.754877666246692759e+00 0)", 0, MPC_RNDNN); ++k;
  p[k] =   49; mpc_init2(n[k],  82); mpc_set_str(n[k], "(-1.768846180306581623745078 6e+00 3.49391918638193279490200 38e-03)", 0, MPC_RNDNN); ++k;
  p[k] =   64; mpc_init2(n[k],  94); mpc_set_str(n[k], "(-1.768844094411812773950119 82187e+00 3.49571967789364227637390 25558e-03)", 0, MPC_RNDNN); ++k;
  p[k] =  117; mpc_init2(n[k], 118); mpc_set_str(n[k], "(-1.768844092775164025862093523045031305e+00 3.49571805780159955197525312189494340 7e-03)", 0, MPC_RNDNN); ++k;
  p[k] =  181; mpc_init2(n[k], 132); mpc_set_str(n[k], "(-1.768844092775157001485004885035344514218 4e+00 3.49571805779999120671448973569976149659 12e-03)", 0, MPC_RNDNN); ++k;
  p[k] =  254; mpc_init2(n[k], 160); mpc_set_str(n[k], "(-1.768844092775156979281952119393989129890207433901 9e+00 3.49571805780022137247427159665955727908552366136 65e-03)", 0, MPC_RNDNN); ++k;
  p[k] =  435; mpc_init2(n[k], 196); mpc_set_str(n[k], "(-1.768844092775156979281945620392209651737031437281978233921 42e+00 3.49571805780022137247145890383652401364436286630462775568058 8e-03)", 0, MPC_RNDNN); ++k;
  p[k] =  508; mpc_init2(n[k], 224); mpc_set_str(n[k], "(-1.768844092775156979281945603594301527954900277030354908568326947927 52e+00 3.49571805780022137247151331544497347465874421145679492139545762726381e-03)", 0, MPC_RNDNN); ++k;
  p[k] =  762; mpc_init2(n[k], 257); mpc_set_str(n[k], "(-1.768844092775156979281945603594301527925590805370612532363237194497526858232e+00 3.49571805780022137247151331544497344939610154942207002297927911118482145794917 2e-03)", 0, MPC_RNDNN); ++k;
  p[k] = 1025; mpc_init2(n[k], 319); mpc_set_str(n[k], "(-1.768844092775156979281945603594301527925653345182949356377147111350931496334054983 4
```

```
      ↪ e+00 ↙
      ↪ 3.4957180578002213724715133154449734497441040991526208736900690746016375909341613367940
      ↪ e-03)", 0, MPC_RNDNN); ++k;
  p[k] = 1533; mpc_init2(n[k], 368); mpc_set_str(n[k], ↙
      ↪ "(-1.7688440927751569792819456035943015279256533451829493563771303761616198557908421271
      ↪ e+00 ↙
      ↪ 3.4957180578002213724715133154449734497441040991526208736843586831742169046155387814490
      ↪ e-03)", 0, MPC_RNDNN); ++k;
55  p[k] = 2050; mpc_init2(n[k], 461); mpc_set_str(n[k], ↙
      ↪ "(-1.7688440927751569792819456035943015279256533451829493563771304367710561424416827170
      ↪ e+00 ↙
      ↪ 3.4957180578002213724715133154449734497441040991526208736844563593757964780333923075502
      ↪ e-03)", 0, MPC_RNDNN); ++k;
  p[k] = 3075; mpc_init2(n[k], 534); mpc_set_str(n[k], ↙
      ↪ "(-1.7688440927751569792819456035943015279256533451829493563771304367710561424416827170
      ↪ e+00 ↙
      ↪ 3.4957180578002213724715133154449734497441040991526208736844563593757964780333923075502
      ↪ e-03)", 0, MPC_RNDNN); ++k;
#endif
#ifdef EXAMPLE_2
  int arm_length = 2;
60  p[k] =     5; mpc_init2(n[k],  60); mpc_set_str(n[k], "(-1.625413725123303739e↙
      ↪ +00 9.723461371658033917 4e-63)", 0, MPC_RNDNN); k++;
  p[k] =   143; mpc_init2(n[k],  86); mpc_set_str(n[k], ↙
      ↪ "(-1.626199074388465482373331 21e+00 2.666509781500971306392159 3e-03)", ↙
      ↪ 0, MPC_RNDNN); k++;
  p[k] =   168; mpc_init2(n[k],  95); mpc_set_str(n[k], ↙
      ↪ "(-1.626198914243429926218539093 8e+00 2.666603227953756744233357833 79e↙
      ↪ -03)", 0, MPC_RNDNN); k++;
  p[k] =   314; mpc_init2(n[k], 116); mpc_set_str(n[k], ↙
      ↪ "(-1.626198914182945117037280024653420 15e+00 ↙
      ↪ 2.666603776560105504607205611379180 04e-03)", 0, MPC_RNDNN); k++;
  p[k] =   482; mpc_init2(n[k], 129); mpc_set_str(n[k], ↙
      ↪ "(-1.626198914182938985538564677089586752005e+00 ↙
      ↪ 2.666603776564300070770030455549097775821e-03)", 0, MPC_RNDNN); k++;
65  p[k] =   660; mpc_init2(n[k], 153); mpc_set_str(n[k], ↙
      ↪ "(-1.626198914182939190773453165714546185773843019 25e+00 ↙
      ↪ 2.666603776564491425228114181241361012372199672 4e-03)", 0, MPC_RNDNN); k↙
      ↪ ++;
  p[k] =  1142; mpc_init2(n[k], 185); mpc_set_str(n[k], ↙
      ↪ "(-1.626198914182939190773360043727218339880780539868125675 5e+00 ↙
      ↪ 2.666603776564491425188858620099903851585925364085597685 38e-03)", 0, ↙
      ↪ MPC_RNDNN); k++;
  p[k] =  1320; mpc_init2(n[k], 209); mpc_set_str(n[k], ↙
      ↪ "(-1.626198914182939190773359851571723918799976100496450303179772746e+00↙
      ↪  2.666603776564491425189798214768480627632121278272245683223110403e-03)↙
      ↪ ", 0, MPC_RNDNN); k++;
  p[k] =  1980; mpc_init2(n[k], 238); mpc_set_str(n[k], ↙
      ↪ "(-1.626198914182939190773359851571723908773386713133162973699947469588474642↙
      ↪ e+00 ↙
      ↪ 2.666603776564491425189798214768507349505965110503748647983181864801564807↙
      ↪ e-03)", 0, MPC_RNDNN); k++;
  p[k] =  2650; mpc_init2(n[k], 291); mpc_set_str(n[k], ↙
      ↪ "(-1.626198914182939190773359851571723909132364258444100570758110615062733924168072455 9
      ↪ e+00 ↙
      ↪ 2.666603776564491425189798214768506976362048864293071426966002111942177462436795239847 9
      ↪ e-03)", 0, MPC_RNDNN); k++;
```

```
    p[k] =  3970; mpc_init2(n[k], 334); mpc_set_str(n[k], ↙
        ↳ "(-1.626198914182939190773359851571723909132364258444100569915041231138513399100273576
        ↳ e+00 ↙
        ↳ 2.666603776564491425189798214768506976362048864293079116127529354390961234549659530469
        ↳ e-03)", 0, MPC_RNDNN); k++;
    p[k] =  5300; mpc_init2(n[k], 415); mpc_set_str(n[k], ↙
        ↳ "(-1.626198914182939190773359851571723909132364258444100569965309496443135875907384945
        ↳ e+00 ↙
        ↳ 2.666603776564491425189798214768506976362048864293079028888760895804370295186782804190
        ↳ e-03)", 0, MPC_RNDNN); k++;
    p[k] =  7950; mpc_init2(n[k], 479); mpc_set_str(n[k], ↙
        ↳ "(-1.626198914182939190773359851571723909132364258444100569965309496443135875907384945
        ↳ e+00 ↙
        ↳ 2.666603776564491425189798214768506976362048864293079028888760895804370295186783561339
        ↳ e-03)", 0, MPC_RNDNN); k++;
    p[k] = 10610; mpc_init2(n[k], 600); mpc_set_str(n[k], ↙
        ↳ "(-1.626198914182939190773359851571723909132364258444100569965309496443135875907384945
        ↳ e+00 ↙
        ↳ 2.666603776564491425189798214768506976362048864293079028888760895804370295186783548292
        ↳ e-03)", 0, MPC_RNDNN); k++;
    p[k] = 15910; mpc_init2(n[k], 695); mpc_set_str(n[k], ↙
        ↳ "(-1.626198914182939190773359851571723909132364258444100569965309496443135875907384945
        ↳ e+00 ↙
        ↳ 2.666603776564491425189798214768506976362048864293079028888760895804370295186783548292
        ↳ e-03)", 0, MPC_RNDNN); k++;
#endif
    const int DEFINED = k;
    mpc_t a, b, c, a2, b2, c2;
    mpc_init2(a, 24);
    mpc_init2(b, 24);
    mpc_init2(c, 24);
    mpc_init2(a2, 24);
    mpc_init2(b2, 24);
    mpc_init2(c2, 24);
    printf("size 1280 720\n# view 53 -0.75 0 1.5\n");
    for (int i = 4; i < COUNT; ++i)
    {
      if ((i & 1) == 0)
      {
        if (DEFINED <= i)
        {
          p[i] = p[i-1] + p[i-3];
          mpc_init2(n[i], mpfr_get_prec(mpc_realref(n[i-2])) * 1.5 - 18);
          mpc_set(n[i], n[i-1], MPC_RNDNN);
          m_r_domain_coord(n[i], n[i], c, p[i-3], p[i-1], newton_steps);
          m_r_nucleus(n[i], n[i], p[i], newton_steps, NCPUS);
        }
        mpc_set(b, a, MPC_RNDNN);
        m_r_domain_coord_of_point(a, n[i], p[i-3], p[i-1]);
        mpc_log(a, a, MPFR_RNDN);
        mpc_div(c, a, b, MPC_RNDNN);
        mpfr_fprintf(stderr, "%d(%d/%d)\t%Re\t%Re\n", p[i], p[i-3], p[i-1], ↙
            ↳ mpc_realref(c), mpc_imagref(c));
        mpc_mul(c, c, a, MPC_RNDNN);
        mpc_exp(c, c, MPC_RNDNN);
      }
      if ((i & 1) == 1)
```

```
    {
      if (DEFINED <= i)
      {
#ifdef EXAMPLE_2
        int misiurewicz_offset = -14 - 10 * (i / 4);
#endif
        p[i] = p[i-1] + p[i-4] + arm_length * p[0];
        mpc_init2(n[i], mpfr_get_prec(mpc_realref(n[i-2])) * 1.5 - 18);
        mpc_set(n[i], n[i-1], MPC_RNDNN);
        m_r_domain_coord(n[i], n[i], c2, p[i-2], p[i-1], newton_steps);
        m_r_misiurewicz(n[i], n[i], p[i] + misiurewicz_offset, p[0], ↙
            ↳ newton_steps);
        m_r_misiurewicz_naive(n[i], n[i], p[i] + misiurewicz_offset, p[0], ↙
            ↳ newton_steps);
        m_r_nucleus(n[i], n[i], p[i], newton_steps, NCPUS);
      }
      mpc_set(b2, a2, MPC_RNDNN);
      m_r_domain_coord_of_point(a2, n[i], p[i-2], p[i-1]);
      mpc_log(a2, a2, MPFR_RNDN);
      mpc_div(c2, a2, b2, MPC_RNDNN);
      mpfr_fprintf(stderr, "%d(%d/%d)\t%Re\t%Re\n", p[i], p[i-2], p[i-1], ↙
          ↳ mpc_realref(c2), mpc_imagref(c2));
      mpc_mul(c2, c2, a2, MPC_RNDNN);
      mpc_exp(c2, c2, MPC_RNDNN);
    }
    mpfr_printf("text %Pd %Re %Re %d\n", mpfr_get_prec(mpc_realref(n[i])), ↙
        ↳ mpc_realref(n[i]), mpc_imagref(n[i]), p[i]);
    fflush(stdout);
  }
  mpc_sub(b, n[COUNT-1], n[COUNT-2], MPC_RNDNN);
  mpc_abs(mpc_realref(a), b, MPFR_RNDN);
  mpfr_mul_d(mpc_realref(a), mpc_realref(a), 2, MPFR_RNDN);
  mpfr_printf("# view %Pd %Re %Re %Re\n", mpfr_get_prec(mpc_realref(n[COUNT-2]))↙
      ↳ , mpc_realref(n[COUNT-2]), mpc_imagref(n[COUNT-2]), mpc_realref(a));
  fflush(stdout);
  mpc_clear(a);
  mpc_clear(b);
  mpc_clear(c);
  mpc_clear(a2);
  mpc_clear(b2);
  mpc_clear(c2);
  for (int i = 0; i < COUNT; ++i)
  {
    mpc_clear(n[i]);
  }
  return 0;
}
```

12 c/bin/m-ejs-tree-morph.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

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

```
#define EXAMPLE_3

#define NCPUS 8

int main()
{
  m_symbolics_init();

#ifdef EXAMPLE_1
  // embedded Julia set is 1_1/2->2_1/2->3_2/5->15_4/7->88
  // https://mathr.co.uk/mandelbrot/web/#!q=1_1/2-%3E2_1/2-%3E3_2/5-%3E15_4/7-%3↙
      ↳ E88
  // the ray goes to the tip of the first morphing in prepartion for the next
  const char *embedded_julia_ray = ↙
      ↳ ".0111000111000110111000110111000111000111000110111000110111000111000111000110111000111
      ↳ ";
  int ray_preperiod = 225;
  int ray_period = 3;
  double _Complex ray_endpoint
    = -1.76525599938987623396492597243303e+00
    +  1.04485517375987067290733632798876e-02 * I;
  int influencing_island_period = 3;
  int embedded_julia_set_period = 88;
  int denominator_of_rotation = 5;
  // tree morphing
  int arm_length = 7;
  // how to scale the view
  double view_size_multiplier = 3600;
#endif

#ifdef EXAMPLE_2
  // embedded Julia set is 1_1/2->2_1/2->3_1/4->4_1/2->8_1/15->116_1/2->119
  // https://mathr.co.uk/mandelbrot/web/#!q=1_1/2-%3E2_1/2-%3E3_1/2-%3E4_1/2-%3↙
      ↳ E8_1/15-%3E116_1/2-%3E119
  // the ray goes to the tip of the first morphing in prepartion for the next
  const char *embedded_julia_ray = ↙
      ↳ ".0111100001111000011110000111100001111000011110000111100001111000011110000111100001111
      ↳ ";
  int ray_preperiod = 287;
  int ray_period = 4;
  double _Complex ray_endpoint
    = -1.94155104121873375917685284273942e+00
    +  1.08484266852462027458038858364597e-04 * I;
  int influencing_island_period = 4;
  int embedded_julia_set_period = 119;
  int denominator_of_rotation = 2;
  // tree morphing
  int arm_length = 9;
  // how to scale the view
  double view_size_multiplier = 32768;
#endif

#ifdef EXAMPLE_3
  // embedded Julia set is 1_1/3->3_1/2->4_11/23->89
  // https://mathr.co.uk/mandelbrot/web/#!q=1_1/3-%3E3_1/2-%3E4_11/23-%3E89
  // the ray goes to the tip of the first morphing in prepartion for the next
```

```
    const char *embedded_julia_ray = ↙
        ↳ ".00110100001101000011010000110100001101000011010000110100001101000011010000110
        ↳ ";
    int ray_preperiod = 195;
    int ray_period = 4;
    double _Complex ray_endpoint
      = -1.60671574317626674138601221936210e-01
      +  1.03694257152534946094999286804216e+00 * I;
    int influencing_island_period = 4;
    int embedded_julia_set_period = 89;
    int denominator_of_rotation = 2;
    // tree morphing
    int arm_length = 1;
    // how to scale the view
    double view_size_multiplier = 8;
#endif

    // how many times to iterate
    int number_of_morphs = 10;

    // ray tracing sharpness
    int ray_sharpness = 8;

    // maximum number of newton steps
    int newton_steps = 64;

    // current and previous periods of the nucleus
    int nperiod = 1;
    int nperiod_old = 1;
    int precision = 53 + 1.5 * nperiod; // number of bits
    mpc_t nucleus;
    mpc_init2(nucleus, precision);

    // current misiurewicz point
    int mpreperiod = 0;
    int mperiod = influencing_island_period;
    mpc_t misiurewicz;
    mpc_init2(misiurewicz, precision);

    // current guess for Misiurewicz point
    mpc_t guess;
    mpc_init2(guess, precision);

    // variables for atom domain coordinates and view size don't need to be so ↙
        ↳ precise
    mpc_t atom_coord;
    mpfr_t atom_coord_size;
    mpc_init2(atom_coord, 53);
    mpfr_init2(atom_coord_size, 53);
    mpfr_set_si(atom_coord_size, 1, MPFR_RNDN);
    mpc_t view_delta;
    mpfr_t view_size;
    mpc_init2(view_delta, 53);
    mpfr_init2(view_size, 53);
    mpfr_set_si(view_size, 1, MPFR_RNDN);

    if (cabs(ray_endpoint) < 0.1)
```

```
  {
    // trace ray towards tip of first level morphed embedded Julia nucleus
    m_binangle binangle;
    m_binangle_init(&binangle);
    m_binangle_from_string(&binangle, embedded_julia_ray);
    mpreperiod = binangle.pre.length;
    mperiod = binangle.per.length;
    mpq_t angle;
    mpq_init(angle);
    m_binangle_to_rational(angle, &binangle);
    m_binangle_clear(&binangle);
    m_r_exray_in *ray = m_r_exray_in_new(angle, ray_sharpness);
    mpq_clear(angle);
    for (int i = 0; i < ray_sharpness * (mpreperiod + mperiod) * 3; ++i) // ↙
        ↳ FIXME check if 3x is always far enough!
    {
      m_r_exray_in_step(ray, newton_steps);
    }
    m_r_exray_in_get(ray, guess); // updates precision too
    m_r_exray_in_delete(ray);
    precision = mpfr_get_prec(mpc_realref(guess));
    mpfr_printf("ray endpoint %dp%d %d %Re %Re\n", mpreperiod, mperiod, ↙
        ↳ precision, mpc_realref(guess), mpc_imagref(guess));
  }
  else
  {
    // have the ray end point already, saves computation time
    mpreperiod = ray_preperiod;
    mperiod = ray_period;
    precision = 100; // FIXME don't hardcode this
    mpfr_set_prec(mpc_realref(guess), precision);
    mpfr_set_prec(mpc_imagref(guess), precision);
    mpc_set_dc(guess, ray_endpoint, MPC_RNDNN);
  }

  // Misiurewicz to tip
  precision = mpfr_get_prec(mpc_realref(guess));
  mpfr_set_prec(mpc_realref(misiurewicz), precision);
  mpfr_set_prec(mpc_imagref(misiurewicz), precision);
  m_r_misiurewicz(misiurewicz, guess, mpreperiod, mperiod, newton_steps);


  // Newton to center
  nperiod_old = embedded_julia_set_period;
  nperiod = embedded_julia_set_period + (arm_length + denominator_of_rotation) *↙
      ↳ influencing_island_period;
  precision = mpfr_get_prec(mpc_realref(guess));
  mpfr_set_prec(mpc_realref(nucleus), precision);
  mpfr_set_prec(mpc_imagref(nucleus), precision);
  m_r_nucleus(nucleus, misiurewicz, nperiod, newton_steps, NCPUS);

  // update period to new target
  nperiod_old = nperiod;
  nperiod = 2 * nperiod + arm_length * influencing_island_period;

  for (int morph = 2; morph <= number_of_morphs + 1; ++morph)
  {
```

```
      // invariant: at loop entry:
      //   nucleus points to a periodic nucleus at the center of a morphing
      //   guess points to the inital guess for misiurewicz finding
170   //   misiurewicz points to the tree-wards tip of the morphing

      // compute view and output
      mpc_sub(view_delta, misiurewicz, nucleus, MPC_RNDNN);
      mpc_abs(view_size, view_delta, MPFR_RNDN);
175   mpfr_mul_d(view_size, view_size, view_size_multiplier, MPFR_RNDN);
      mpfr_printf
        ( "size 1280 720\n"
          "view %d %Re %Re %Re\n"
          "text %d %Re %Re %d\n"
180       "text %d %Re %Re %dp%d\n"
          "text %d %Re %Re GUESS\n"
          "\n"
        , precision, mpc_realref(nucleus), mpc_imagref(nucleus), view_size
        , precision, mpc_realref(nucleus), mpc_imagref(nucleus), nperiod_old
185     , precision, mpc_realref(misiurewicz), mpc_imagref(misiurewicz), ↙
            ↳ mpreperiod, mperiod
        , precision, mpc_realref(guess), mpc_imagref(guess)
        );
      fflush(stdout);
      if (morph == number_of_morphs + 1)
190   {
        break;
      }

      // calculate atom domain coord of tip
195   mpc_t z, zp, zq;
      mpfr_t mz2, z2;
      mpc_init2(z, precision);
      mpc_init2(zp, precision);
      mpc_init2(zq, precision);
200   mpfr_init2(mz2, precision);
      mpfr_init2(z2, precision);
      mpc_set_si(z, 0, MPC_RNDNN);
      mpfr_set_d(mz2, 1.0 / 0.0, MPFR_RNDN);
      for (int i = 1; i <= nperiod_old; ++i)
205   {
        mpc_sqr(z, z, MPC_RNDNN);
        mpc_add(z, z, misiurewicz, MPC_RNDNN);
        mpc_norm(z2, z, MPFR_RNDN);
        if (mpfr_less_p(z2, mz2))
210     {
          mpfr_set(mz2, z2, MPFR_RNDN);
          mpc_set(zq, zp, MPC_RNDNN);
          mpc_set(zp, z, MPC_RNDNN);
        }
215   }
      mpc_div(atom_coord, zp, zq, MPC_RNDNN);
      mpc_clear(z);
      mpc_clear(zp);

220   // scale atom coord by **1.5 FIXME make this a settable variable?
      mpc_abs(atom_coord_size, atom_coord, MPC_RNDNN);
      mpc_div_fr(atom_coord, atom_coord, atom_coord_size, MPC_RNDNN);
```

```
    mpfr_sqrt(atom_coord_size, atom_coord_size, MPFR_RNDN);
    mpfr_pow_si(atom_coord_size, atom_coord_size, 3, MPFR_RNDN);
    mpc_mul_fr(atom_coord, atom_coord, atom_coord_size, MPC_RNDNN);

#ifdef EXAMPLE_3
    // FIXME HACK for example 3 - the guess is too off rotationally otherwise
    mpfr_swap(mpc_realref(atom_coord), mpc_imagref(atom_coord));
#endif

    // nucleus is in arm near misiurewicz tip
    m_r_nucleus(nucleus, misiurewicz, nperiod, newton_steps, NCPUS);

    // guess is atom coord relative to new nucleus
    m_r_domain_coord(guess, nucleus, atom_coord, nperiod_old, nperiod, ↙
        ↳ newton_steps);

    // update period to new target
    nperiod_old = nperiod;
    nperiod = 2 * nperiod + arm_length * influencing_island_period;

    // set precision
    precision = 53 - 3 * mpfr_get_exp(view_size);
    mpfr_prec_round(mpc_realref(nucleus), precision, MPFR_RNDN);
    mpfr_prec_round(mpc_imagref(nucleus), precision, MPFR_RNDN);
    mpfr_prec_round(mpc_realref(guess), precision, MPFR_RNDN);
    mpfr_prec_round(mpc_imagref(guess), precision, MPFR_RNDN);
    mpfr_prec_round(mpc_realref(misiurewicz), precision, MPFR_RNDN);
    mpfr_prec_round(mpc_imagref(misiurewicz), precision, MPFR_RNDN);

    // misiurewicz is guess refined to new tip
    int d = influencing_island_period - 1 + arm_length * ↙
        ↳ influencing_island_period * (morph + 1);
    mpreperiod = nperiod - d;
    mperiod = influencing_island_period;
    m_r_misiurewicz(misiurewicz, guess, mpreperiod, mperiod, newton_steps);

  }

  m_symbolics_exit();
  return 0;
}
```

13 c/bin/m-equipotential.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2019 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

#include <stdio.h>
#include <mandelbrot-numerics.h>
#include <mandelbrot-symbolics.h>
#include "m-util.h"

static void usage(const char *progname) {
  fprintf
    ( stderr
    , "usage: %s precision count sharpness\n"
```

```
    , progname
    );
}

extern int main(int argc, char **argv) {
  if (argc != 4) {
    usage(argv[0]);
    return 1;
  }
  bool native = true;
  int bits = 0;
  int count = 0;
  int sharpness = 0;
  int newton = 64;
  if (! arg_precision(argv[1], &native, &bits)) { return 1; }
  if (! arg_int(argv[2], &count)) { return 1; }
  if (! arg_int(argv[3], &sharpness)) { return 1; }
  int retval = 0;
  if (native)
  {
    mpq_t angle;
    mpq_init(angle);
    mpq_set_si(angle, 0, 1);
    mpq_canonicalize(angle);
    double _Complex c = m_d_exray_in_do(angle, sharpness, sharpness * count, ↙
        ↳ newton);
    mpq_clear(angle);
    m_d_equipotential *ray = m_d_equipotential_new(c, sharpness, count);
    int64_t steps = ((int64_t) sharpness) << count;
    for (int64_t i = 0; i < steps; ++i)
    {
      printf("%.16e %.16e\n", creal(c), cimag(c));
      if (m_stepped != m_d_equipotential_step(ray, newton))
      {
        retval = 1;
        break;
      }
      c = m_d_equipotential_get(ray);
    }
    m_d_equipotential_delete(ray);
  }
  else
  {
    fprintf(stderr, "non-native equipotential not yet implemented\n");
    retval = 1;
  }
  return retval;
}
```

14 c/bin/m-exray-in.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

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

```
#include <mandelbrot-symbolics.h>
#include "m-util.h"

static void usage(const char *progname) {
  fprintf
    ( stderr
    , "usage: %s precision angle sharpness maxsteps\n"
    , progname
    );
}

extern int main(int argc, char **argv) {
  if (argc != 5) {
    usage(argv[0]);
    return 1;
  }
  bool native = true;
  int bits = 0;
  if (! arg_precision(argv[1], &native, &bits)) { return 1; }
  mpq_t angle;
  mpq_init(angle);
  int sharpness = 0;
  int maxsteps = 0;
  if (! arg_rational(argv[2], angle)) {
    m_binangle ba;
    m_binangle_init(&ba);
    const char *s = m_binangle_from_string(&ba, argv[2]);
    if ((!s) || *s) return 1;
    m_binangle_to_rational(angle, &ba);
    m_binangle_clear(&ba);
  }
  if (! arg_int(argv[3], &sharpness)) { mpq_clear(angle); return 1; }
  if (! arg_int(argv[4], &maxsteps)) { mpq_clear(angle); return 1; }
  int retval = 0;
  if (native) {
    m_d_exray_in *ray = m_d_exray_in_new(angle, sharpness);
    if (! ray) { mpq_clear(angle); return 1; }
    for (int i = 0; i < maxsteps; ++i) {
      complex double c = m_d_exray_in_get(ray);
      printf("%.16e %.16e\n", creal(c), cimag(c));
      if (m_stepped != m_d_exray_in_step(ray, 8)) {
        retval = 1;
        break;
      }
    }
    m_d_exray_in_delete(ray);
  } else {
    m_r_exray_in *ray = m_r_exray_in_new(angle, sharpness);
    if (! ray) { mpq_clear(angle); return 1; }
    mpc_t c;
    mpc_init2(c, 53);
    for (int i = 0; i < maxsteps; ++i) {
      m_r_exray_in_get(ray, c);
      mpfr_printf("%Re %Re\n", mpc_realref(c), mpc_imagref(c));
      if (m_stepped != m_r_exray_in_step(ray, 8)) {
        retval = 1;
        break;
```

```
      }
      // prec
      mpfr_prec_t precr, preci, prec;
      mpc_get_prec2(&precr, &preci, c);
      prec = precr > preci ? precr : preci;
      if (prec > bits) {
        break;
      }
    }
    m_r_exray_in_delete(ray);
  }
  mpq_clear(angle);
  return retval;
}
```

15 c/bin/m-exray-out.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

#include <stdio.h>
#include <mandelbrot-numerics.h>
#include "m-util.h"

static void usage(const char *progname) {
  fprintf
    ( stderr
    , "usage: %s precision c-re c-im sharpness maxdwell preperiod period\n"
    , progname
    );
}

extern int main(int argc, char **argv) {
  if (argc != 8) {
    usage(argv[0]);
    return 1;
  }
  bool native = true;
  int bits = 0;
  if (! arg_precision(argv[1], &native, &bits)) { return 1; }
  if (native) {
    double cre = 0;
    double cim = 0;
    int sharpness = 0;
    int maxdwell = 0;
    int preperiod = 0;
    int period = 0;
    if (! arg_double(argv[2], &cre)) { return 1; }
    if (! arg_double(argv[3], &cim)) { return 1; }
    if (! arg_int(argv[4], &sharpness)) { return 1; }
    if (! arg_int(argv[5], &maxdwell)) { return 1; }
    if (! arg_int(argv[6], &preperiod)) { return 1; }
    if (! arg_int(argv[7], &period)) { return 1; }
    if (! (preperiod >= 0)) { return 1; }
    if (! (period >= 0)) { return 1; }
    if (preperiod > 0 && ! (period > 0)) { return 1; }
```

```
    m_d_exray_out *ray = m_d_exray_out_new(cre + I * cim, sharpness, maxdwell);
    if (! ray) { return 1; }
    char *sbits = malloc(maxdwell + 2);
    if (! sbits) { m_d_exray_out_delete(ray); return 1; }
    int n = 0;
    do {
      complex double c = m_d_exray_out_get(ray);
      printf("%.16e %.16e", creal(c), cimag(c));
      if (m_d_exray_out_have_bit(ray)) {
        int bit = m_d_exray_out_get_bit(ray);
        sbits[n++] = '0' + bit;
        printf(" %d\n", bit);
      } else {
        printf("\n");
      }
    } while (m_stepped == m_d_exray_out_step(ray));
    sbits[n] = 0;
    m_d_exray_out_delete(ray);
    if (preperiod + period > 0) {
      printf("\n.");
      for (int i = 0; i < preperiod && 0 <= n - 1 - i; ++i) {
        putchar(sbits[n - 1 - i]);
      }
      putchar('(');
      for (int i = preperiod; i < preperiod + period && 0 <= n - 1 - i; ++i) {
        putchar(sbits[n - 1 - i]);
      }
      putchar(')');
      putchar('\n');
    }
    free(sbits);
    return 0;
  } else {
    mpc_t c;
    int sharpness = 0;
    int maxdwell = 0;
    int preperiod = 0;
    int period = 0;
    mpc_init2(c, bits);
    if (! arg_mpc(argv[2], argv[3], c)) { return 1; }
    if (! arg_int(argv[4], &sharpness)) { return 1; }
    if (! arg_int(argv[5], &maxdwell)) { return 1; }
    if (! arg_int(argv[6], &preperiod)) { return 1; }
    if (! arg_int(argv[7], &period)) { return 1; }
    if (! (preperiod >= 0)) { return 1; }
    if (! (period >= 0)) { return 1; }
    if (preperiod > 0 && ! (period > 0)) { return 1; }
    m_r_exray_out *ray = m_r_exray_out_new(c, sharpness, maxdwell, 65536);
    if (! ray) { return 1; }
    char *sbits = malloc(maxdwell + 2);
    if (! sbits) { m_r_exray_out_delete(ray); return 1; }
    int n = 0;
    do {
      m_r_exray_out_get(ray, c);
      mpfr_printf("%Re %Re", mpc_realref(c), mpc_imagref(c));
      if (m_r_exray_out_have_bit(ray)) {
        int bit = m_r_exray_out_get_bit(ray);
```

```
        sbits[n++] = '0' + bit;
        printf(" %d\n", bit);
      } else {
        printf("\n");
      }
    } while (m_stepped == m_r_exray_out_step(ray));
    sbits[n] = 0;
    m_r_exray_out_delete(ray);
    if (preperiod + period > 0) {
      printf("\n.");
      for (int i = 0; i < preperiod && 0 <= n - 1 - i; ++i) {
        putchar(sbits[n - 1 - i]);
      }
      putchar('(');
      for (int i = preperiod; i < preperiod + period && 0 <= n - 1 - i; ++i) {
        putchar(sbits[n - 1 - i]);
      }
      putchar(')');
      putchar('\n');
    }
    free(sbits);
    return 0;
  }
  return 1;
}
```

16 c/bin/m-exray-out-perturbed.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2019 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

#include <stdio.h>
#include <mandelbrot-numerics.h>
#include "m-util.h"

static void usage(const char *progname) {
  fprintf
    ( stderr
    , "usage: %s precision nucleus-re nucleus-im nucleus-period c-re c-im ↙
      ↳ sharpness maxdwell preperiod period\n"
    , progname
    );
}

extern int main(int argc, char **argv) {
  if (argc != 11) {
    usage(argv[0]);
    return 1;
  }
  bool native = true;
  int bits = 0;
  if (! arg_precision(argv[1], &native, &bits)) { return 1; }
  if (native) {
    fprintf(stderr, "no native\n");
    return 1;
  } else {
```

```
    mpc_t nucleus, endpoint;
    int sharpness = 0;
    int maxdwell = 0;
    int preperiod = 0;
    int period = 0;
    int nperiod = 0;
    mpc_init2(nucleus, bits);
    mpc_init2(endpoint, bits);
    if (! arg_mpc(argv[2], argv[3], nucleus)) { return 1; }
    if (! arg_int(argv[4], &nperiod)) { return 1; }
    if (! arg_mpc(argv[5], argv[6], endpoint)) { return 1; }
    if (! arg_int(argv[7], &sharpness)) { return 1; }
    if (! arg_int(argv[8], &maxdwell)) { return 1; }
    if (! arg_int(argv[9], &preperiod)) { return 1; }
    if (! arg_int(argv[10], &period)) { return 1; }
    if (! (preperiod >= 0)) { return 1; }
    if (! (nperiod > 0)) { return 1; }
    if (! (period >= 0)) { return 1; }
    if (preperiod > 0 && ! (period > 0)) { return 1; }
    m_r_exray_out_perturbed *ray = m_r_exray_out_perturbed_new(nucleus, nperiod,↙
      ↳ endpoint, sharpness, maxdwell, 65536);
    if (! ray) { return 1; }
    char *sbits = malloc(maxdwell + 2);
    if (! sbits) { m_r_exray_out_perturbed_delete(ray); return 1; }
    int n = 0;
    do {
      double _Complex c = m_r_exray_out_perturbed_get(ray);
      printf("%.18e %.18e", creal(c), cimag(c));
      if (m_r_exray_out_perturbed_have_bit(ray)) {
        int bit = m_r_exray_out_perturbed_get_bit(ray);
        sbits[n++] = '0' + bit;
        printf(" %d\n", bit);
      } else {
        printf("\n");
      }
    } while (m_stepped == m_r_exray_out_perturbed_step(ray));
    sbits[n] = 0;
    m_r_exray_out_perturbed_delete(ray);
    if (preperiod + period > 0) {
      printf("\n.");
      for (int i = 0; i < preperiod && 0 <= n - 1 - i; ++i) {
        putchar(sbits[n - 1 - i]);
      }
      putchar('(');
      for (int i = preperiod; i < preperiod + period && 0 <= n - 1 - i; ++i) {
        putchar(sbits[n - 1 - i]);
      }
      putchar(')');
      putchar('\n');
    }
    free(sbits);
    return 0;
  }
  return 1;
}
```

17 c/bin/m-feature-database.c

```
#include <assert.h>
#include <stdio.h>
#include <mandelbrot-numerics.h>
#include <mandelbrot-symbolics.h>
#include "m-util.h"

int main(int argc, char **argv)
{
  if (! (argc == 3))
  {
    fprintf(stderr, "usage: %s preperiod period\n", argv[0]);
    return 1;
  }
  int preperiod = 0;
  int period = 1;
  int sharpness = 16;
  int maxsteps = 100;
  int ncpus = 0;
  arg_int(argv[1], &preperiod);
  assert(preperiod >= 0);
  assert(preperiod < 64);
  arg_int(argv[2], &period);
  assert(period > 0);
  assert(period < 64);
  mpfr_prec_t precision = 4 * (preperiod + period);
  int depth = (precision + 16) * sharpness;
  int digits = log10(2) * precision;
  m_binangle angle;
  m_binangle_init(&angle);
  mpq_t q;
  mpq_init(q);
  mpc_t endpoint, root, output, delta;
  mpc_init2(endpoint, precision + 16);
  mpc_init2(root, precision + 16);
  mpc_init2(output, precision);
  mpc_init2(delta, 16);
  mpfr_t distance;
  mpfr_init2(distance, 16);
  for (uint64_t pre = 0; pre < ((uint64_t)1) << preperiod; ++pre)
  {
    for (uint64_t per = 0; per < ((uint64_t)1) << period; ++per)
    {
      mpz_set_ui(angle.pre.bits, pre);
      angle.pre.length = preperiod;
      mpz_set_ui(angle.per.bits, per);
      angle.per.length = period;
      m_binangle_to_rational(q, &angle);
      m_binangle_from_rational(&angle, q);
      if (angle.pre.length == preperiod && angle.per.length == period &&
          mpz_get_ui(angle.pre.bits) == pre && mpz_get_ui(angle.per.bits) == per
              && &&
          mpq_cmp_ui(q, 1, 2) <= 0)
      {
        m_r_exray_in *ray = m_r_exray_in_new(q, sharpness);
        m_newton r;
```

```
55          for (int i = 0; i < depth; ++i)
            {
              r = m_r_exray_in_step(ray, maxsteps);
              if (r == m_failed)
              {
60              fprintf(stderr, "FAIL %lu %lu %d\n", pre, per, i);
                abort();
              }
            }
            m_r_exray_in_get(ray, endpoint);
65          mpc_set(root, endpoint, MPC_RNDNN);
            if (preperiod == 0)
            {
              r = m_r_nucleus(root, root, period, maxsteps, ncpus);
            }
70          else
            {
              r = m_r_misiurewicz(root, root, preperiod + 1, period, maxsteps);
              if (r != m_failed)
              {
75              r = m_r_misiurewicz_naive(root, root, preperiod + 1, period, ↙
                    ↳ maxsteps);
              }
            }
            if (r == m_failed)
            {
80            fprintf(stderr, "FAIL %lu %lu\n", pre, per);
              abort();
            }
            mpc_sub(delta, root, endpoint, MPC_RNDNN);
            mpc_norm(distance, delta, MPFR_RNDN);
85          mpfr_exp_t e = -mpfr_get_exp(distance);
            if (1 || (e > 2 * precision))
            {
              mpc_set(output, root, MPC_RNDNN);
              mpfr_abs(mpc_imagref(output), mpc_imagref(output), MPFR_RNDN);
90            char *s = malloc(m_binangle_strlen(&angle));
              m_binangle_to_string(s, &angle);
              mpfr_printf("%s\t%Qd\t%.*Rf\t%.*Rf\n", s, q, digits, mpc_realref(↙
                  ↳ output), digits, mpc_imagref(output));
              free(s);
              fflush(stdout);
95          }
            else
            {
              fprintf(stderr, "FAIL %lu %lu e=%ld\n", pre, per, e);
              abort();
100         }
          }
        }
      }
      return 0;
105 }
```

18 c/bin/m-feigenbaum3.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
```

```
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

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

#include <mandelbrot-numerics.h>

#define twopi 6.283185307179586

int main(int argc, char **argv) {
  (void) argc;
  (void) argv;
  if (argc != 4) return 1;
  complex double location = atof(argv[1]) + I * atof(argv[2]);
  int periodFactor = atoi(argv[3]);
  int period = 1;
  complex double nucleus = 0;
  complex double size = 1;
;
  for (int depth = 0; depth < 12; ++depth) {
    size = m_d_size(nucleus, period);
    complex double estimate = nucleus + location * size;
    period *= periodFactor;
    m_d_nucleus(&nucleus, estimate, period, 64);
  }
  complex double c0 = nucleus;
  nucleus = 0;
  size = 1;
  period = 1;
  for (int depth = 0; depth < 12; ++depth) {
    size = m_d_size(nucleus, period);
    complex double d0 = nucleus - c0;
    complex double estimate = nucleus + location * size;
    period *= periodFactor;
    m_d_nucleus(&nucleus, estimate, period, 64);
    complex double d1 = nucleus - c0;
    complex double r = d0 / d1;
    printf
      ( "#preset \nC = %.18e,%.18e\nZ = %.18e,%.18e\nT = %d\n#endpreset\n"
      , creal(c0), cimag(c0), creal(r), cimag(r), (int)round(25 * log10(cabs(r))↙
        ↳ )
      );
    fflush(stdout);
  }
  return 0;
}
```

19 c/bin/m-feigenbaum.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

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

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

#define twopi 6.283185307179586

int main(int argc, char **argv) {
  (void) argc;
  (void) argv;
  int count = 32;
  int period = 2;
  double size = 0.5;
  complex double nucleus = -1;
  printf("%.18e\n", -1.0);
  for (int depth = 1; depth < count; ++depth) {
    period *= 2;
    complex double estimate = nucleus - 1.25 * (size / 2);
    m_d_nucleus(&nucleus, estimate, period, 64);
    size = cabs(m_d_size(nucleus, period));
    printf("%.18e\n", creal(nucleus));
    fflush(stdout);
  }
  return 0;
}
```

20 c/bin/m-fibonacci-phi.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

#include <stdio.h>
#include <gmp.h>
#include <mandelbrot-numerics.h>
#include "m-util.h"

int main(int argc, char **argv)
{
  (void) argc;
  (void) argv;
  mpq_t a, b, c;
  mpq_init(a);
  mpq_init(b);
  mpq_init(c);
  mpq_set_si(a, 0, 1);
  mpq_set_si(b, 1, 1);
  double prev_size = 0.0/0.0;
  for (int i = 0; i < 24; ++i)
  {
    mpz_add(mpq_numref(c), mpq_numref(a), mpq_numref(b));
    mpz_add(mpq_denref(c), mpq_denref(a), mpq_denref(b));
    mpq_canonicalize(c);
    complex double z = 0, bond = 0, nucleus = 0;
    m_d_interior(&z, &bond, 0, 0, cexp(I * twopi * mpq_get_d(c)), 1, 64);
    int period = mpz_get_si(mpq_denref(c));
    m_d_nucleus(&nucleus, bond, period, 64);
    double size = cabs(m_d_size(nucleus, period));
    printf
      ( "%d/%d\t%.18e\t%.18e\n"
```

```
      , (int) mpz_get_si(mpq_numref(c))
      , period
      , size
      , sqrt(prev_size / size)
      );
    fflush(stdout);
    mpq_set(a, b);
    mpq_set(b, c);
    prev_size = size;
  }
  mpq_clear(a);
  mpq_clear(b);
  mpq_clear(c);
  return 0;
}
```

21 c/bin/m-furcation.c

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

#include <mandelbrot-numerics.h>

#include "m-util.h"

static void usage(const char *progname) {
  fprintf
    ( stderr
    , "usage: %s angle... | ffmpeg -r 10 -i - output.gif\n"
    , progname
    );
}

extern int main(int argc, char **argv) {
  if (! (argc > 1)) {
    usage(argv[0]);
    return 1;
  }
  const int maxsteps = 64;
  const int sharpness = 64;
  mpq_t q;
  mpq_init(q);
  double dperiod = 1;
  for (int arg = 1; 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 = 2 * sharpness * (argc - 1);
  int width  = height;
  unsigned char *ppm = malloc(width * height * 3);
  for (int angle = 0; angle < 360; angle += 5)
  {
```

```
    memset(ppm, 0, width * height * 3);
    int period = 1;
    double _Complex c = 0;
    double _Complex z = 0;
    double _Complex dc = 0;
    double _Complex dz = 0;
    double _Complex rotation = cexp(I * twopi * angle / 360.0);
    double _Complex left  = cexp(I * twopi *  2.0 / 360) * rotation;
    double _Complex right = cexp(I * twopi * -2.0 / 360) * rotation;
    int x = 0;
    for (int arg = 1; arg < argc; ++arg)
    {
      arg_rational(argv[arg], q);
      m_d_nucleus(&c, c, period, maxsteps);
      z = 0;
      for (int step = 0; step < sharpness; ++step)
      {
        int y = (2 * arg - 1) * sharpness - 1 - step;
        m_d_interior(&z, &c, z, c, (step + 0.5) / sharpness, period, maxsteps);
        double _Complex w = z;
        for (int att = 0; att < period; ++att)
        {
          x = cimag(right * w) / cabs(right * w - 10) * 4 * width + width / 2;
          if (x < 0) x = 0;
          if (x >= width) x = width - 1;
          ppm[3 * width * y + 3 * x + 0] = 255;
          x = cimag(left * w) / cabs(left * w - 10) * 4 * width + width / 2;
          if (x < 0) x = 0;
          if (x >= width) x = width - 1;
          ppm[3 * width * y + 3 * x + 1] = 255;
          ppm[3 * width * y + 3 * x + 2] = 255;
          w = w * w + c;
        }
      }
      double t = twopi * mpq_get_d(q);
      double _Complex v = cexp(I * t);
      m_d_nucleus(&c, c, period, maxsteps);
      z = 0;
      for (int step = 0; step < sharpness; ++step)
      {
        int y = (2 * arg - 1) * sharpness + step;
        double _Complex z1, c1;
        m_d_interior(&z1, &c1, z, c, (step + 0.5) / sharpness * v, period, ↙
            ↳ maxsteps);
        dc = c1 - c;
        dz = z1 - z;
        c = c1;
        z = z1;
        double _Complex w = z;
        for (int att = 0; att < period; ++att)
        {
          x = cimag(right * w) / cabs(right * w - 10) * 4 * width + width / 2;
          if (x < 0) x = 0;
          if (x >= width) x = width - 1;
          ppm[3 * width * y + 3 * x + 0] = 255;
          x = cimag(left * w) / cabs(left * w - 10) * 4 * width + width / 2;
          if (x < 0) x = 0;
```

```
            if (x >= width) x = width - 1;
            ppm[3 * width * y + 3 * x + 1] = 255;
            ppm[3 * width * y + 3 * x + 2] = 255;
            w = w * w + c;
          }
          ++y;
        }
        int p = mpz_get_ui(mpq_denref(q));
        double f = 0.5 * (1 + (period == 1 ? sin(0.5 * p) : 1.0) / (p * p));
        c += dc * f;
        z += dz * f;
        period *= p;
      }
      printf("P6\n%d %d\n255\n", width, height);
      fwrite(ppm, width * height * 3, 1, stdout);
      fflush(stdout);
    }
    mpq_clear(q);
    free(ppm);
    return 0;
}
```

22 c/bin/m-interior.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

#include <stdio.h>
#include <mandelbrot-numerics.h>
#include "m-util.h"

static void usage(const char *progname) {
  fprintf
    ( stderr
    , "usage: %s precision z-guess-re z-guess-im c-guess-re c-guess-im interior-r interior-t period maxsteps\n"
    , progname
    );
}

extern int main(int argc, char **argv) {
  if (argc != 10) {
    usage(argv[0]);
    return 1;
  }
  bool native = true;
  int bits = 0;
  if (! arg_precision(argv[1], &native, &bits)) { return 1; }
  if (native) {
    double zre = 0;
    double zim = 0;
    double cre = 0;
    double cim = 0;
    double ir = 0;
    double it = 0;
    int period = 0;
```

```
    int maxsteps = 0;
    if (! arg_double(argv[2], &zre)) { return 1; }
    if (! arg_double(argv[3], &zim)) { return 1; }
    if (! arg_double(argv[4], &cre)) { return 1; }
    if (! arg_double(argv[5], &cim)) { return 1; }
    if (! arg_double(argv[6], &ir)) { return 1; }
    if (! arg_double(argv[7], &it)) { return 1; }
    if (! arg_int(argv[8], &period)) { return 1; }
    if (! arg_int(argv[9], &maxsteps)) { return 1; }
    complex double z = 0;
    complex double c = 0;
    m_d_interior(&z, &c, zre + I * zim, cre + I * cim, ir * cexp(I * twopi * it)
      , period, maxsteps);
    printf("%.16e %.16e %.16e %.16e\n", creal(z), cimag(z), creal(c), cimag(c));
    return 0;
  } else {
    mpc_t z, c, i, interior;
    mpfr_t p;
    mpc_init2(z, bits);
    mpc_init2(c, bits);
    mpc_init2(i, bits);
    mpc_init2(interior, bits);
    mpfr_init2(p, bits);
    int period = 0;
    int maxsteps = 0;
    if (! arg_mpc(argv[2], argv[3], z)) { return 1; }
    if (! arg_mpc(argv[4], argv[5], c)) { return 1; }
    if (! arg_mpc(argv[6], argv[7], i)) { return 1; }
    if (! arg_int(argv[8], &period)) { return 1; }
    if (! arg_int(argv[9], &maxsteps)) { return 1; }
    // interior = ir * cexp(I * twopi * it);
    mpfr_const_pi(p, MPFR_RNDN);
    mpfr_mul(mpc_imagref(i), mpc_imagref(i), p, MPFR_RNDN);
    mpfr_mul_2si(mpc_imagref(i), mpc_imagref(i), 1, MPFR_RNDN);
    mpfr_sin_cos(mpc_imagref(interior), mpc_realref(interior), mpc_imagref(i),
      MPFR_RNDN);
    mpc_mul_fr(interior, interior, mpc_realref(i), MPC_RNDNN);
    m_r_interior(z, c, z, c, interior, period, maxsteps);
    mpfr_printf("%Re %Re %Re %Re\n", mpc_realref(z), mpc_imagref(z), mpc_realref
      (c), mpc_imagref(c));
    mpc_clear(z);
    mpc_clear(c);
    mpc_clear(i);
    mpc_clear(interior);
    mpfr_clear(p);
    return 0;
  }
  return 1;
}
```

23 c/bin/m-misiurewicz.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

#include <stdio.h>
```

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

static void usage(const char *progname) {
  fprintf
    ( stderr
    , "usage: %s precision guess-re guess-im preperiod period maxsteps\n"
    , progname
    );
}

extern int main(int argc, char **argv) {
  if (argc != 7) {
    usage(argv[0]);
    return 1;
  }
  bool native = true;
  int bits = 0;
  if (! arg_precision(argv[1], &native, &bits)) { return 1; }
  if (native) {
    double cre = 0;
    double cim = 0;
    int preperiod = 0;
    int period = 0;
    int maxsteps = 0;
    if (! arg_double(argv[2], &cre)) { return 1; }
    if (! arg_double(argv[3], &cim)) { return 1; }
    if (! arg_int(argv[4], &preperiod)) { return 1; }
    if (! arg_int(argv[5], &period)) { return 1; }
    if (! arg_int(argv[6], &maxsteps)) { return 1; }
    complex double c = 0;
    m_d_misiurewicz(&c, cre + I * cim, preperiod, period, maxsteps);
    m_d_misiurewicz_naive(&c, c, preperiod, period, maxsteps);
    printf("%.16e %.16e\n", creal(c), cimag(c));
    return 0;
  } else {
    mpc_t c_guess;
    int preperiod = 0;
    int period = 0;
    int maxsteps = 0;
    mpc_init2(c_guess, bits);
    if (! arg_mpc(argv[2], argv[3], c_guess)) { return 1; }
    if (! arg_int(argv[4], &preperiod)) { return 1; }
    if (! arg_int(argv[5], &period)) { return 1; }
    if (! arg_int(argv[6], &maxsteps)) { return 1; }
    mpc_t c_out;
    mpc_init2(c_out, bits);
    m_r_misiurewicz(c_out, c_guess, preperiod, period, maxsteps);
    m_r_misiurewicz_naive(c_out, c_out, preperiod, period, maxsteps);
    mpfr_printf("%Re %Re\n", mpc_realref(c_out), mpc_imagref(c_out));
    mpc_clear(c_out);
    mpc_clear(c_guess);
    return 0;
  }
  return 1;
}
```

24 c/bin/m-nearest-roots.c

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

#define NCPUS 8

int main()
{
  mpfr_t distance;
  mpfr_init2(distance, 53);
  for (int period = 4; period <= 65536; period <<= 1)
  {
    mpc_t nucleus[3];
    mpc_t delta;
    mpc_init2(nucleus[0], 4 * period);
    mpc_init2(nucleus[1], 4 * period);
    mpc_init2(nucleus[2], 4 * period);
    mpc_init2(delta, 4 * period);
    mpc_set_si(nucleus[0], -2, MPC_RNDNN);
    mpc_set_si(nucleus[1], -2, MPC_RNDNN);
    m_r_nucleus(nucleus[0], nucleus[0], period, 64, NCPUS);
    m_r_nucleus(nucleus[1], nucleus[1], period + 1, 64, NCPUS);
    mpc_sub(delta, nucleus[0], nucleus[1], MPC_RNDNN);
    mpc_add(nucleus[2], nucleus[0], delta, MPC_RNDNN);
    m_r_nucleus(nucleus[2], nucleus[2], period + 1, 64, NCPUS);
    mpc_sub(delta, nucleus[1], nucleus[2], MPC_RNDNN);
    mpc_abs(distance, delta, MPFR_RNDN);
    mpfr_log2(distance, distance, MPFR_RNDN);
    mpfr_div_si(distance, distance, period + 1, MPFR_RNDN);
    mpfr_printf("%6d\t%Re\n", period + 1, distance);
    mpc_clear(nucleus[0]);
    mpc_clear(nucleus[1]);
    mpc_clear(nucleus[2]);
    mpc_clear(delta);
  }
  return 0;
}
```

25 c/bin/m-nucleus.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

#include <stdio.h>
#include <mandelbrot-numerics.h>
#include "m-util.h"

static void usage(const char *progname) {
  fprintf
    ( stderr
    , "usage:\n"
      "%s precision guess-re guess-im period maxsteps ncpus\n"
      "%s precision period maxsteps ncpus <<EOF\n"
      "guess-re\nguess-im\nEOF\n"
    , progname, progname
```

```
    );
}

extern int main(int argc, char **argv) {
  if (argc != 5 && argc != 7) {
    usage(argv[0]);
    return 1;
  }
  bool native = true;
  int bits = 0;
  if (! arg_precision(argv[1], &native, &bits)) { return 1; }
  if (native) {
    double cre = 0;
    double cim = 0;
    int period = 0;
    int maxsteps = 0;
    int ncpus = 0;
    if (argc == 5)
    {
      if (! stdin_double(&cre)) { return 1; }
      if (! stdin_double(&cim)) { return 1; }
      if (! arg_int(argv[2], &period)) { return 1; }
      if (! arg_int(argv[3], &maxsteps)) { return 1; }
      if (! arg_int(argv[4], &ncpus)) { return 1; }
    }
    else
    {
      if (! arg_double(argv[2], &cre)) { return 1; }
      if (! arg_double(argv[3], &cim)) { return 1; }
      if (! arg_int(argv[4], &period)) { return 1; }
      if (! arg_int(argv[5], &maxsteps)) { return 1; }
      if (! arg_int(argv[6], &ncpus)) { return 1; }
    }
    complex double c = 0;
    m_d_nucleus(&c, cre + I * cim, period, maxsteps);
    printf("%.16e %.16e\n", creal(c), cimag(c));
    return 0;
  } else {
    mpc_t c_guess;
    int period = 0;
    int maxsteps = 0;
    int ncpus = 0;
    mpc_init2(c_guess, bits);
    if (argc == 5)
    {
      if (! stdin_mpfr(mpc_realref(c_guess))) { return 1; }
      if (! stdin_mpfr(mpc_imagref(c_guess))) { return 1; }
      if (! arg_int(argv[2], &period)) { return 1; }
      if (! arg_int(argv[3], &maxsteps)) { return 1; }
      if (! arg_int(argv[4], &ncpus)) { return 1; }
    }
    else
    {
      if (! arg_mpc(argv[2], argv[3], c_guess)) { return 1; }
      if (! arg_int(argv[4], &period)) { return 1; }
      if (! arg_int(argv[5], &maxsteps)) { return 1; }
      if (! arg_int(argv[6], &ncpus)) { return 1; }
```

```
    }
    mpc_t c_out;
    mpc_init2(c_out, bits);
    m_r_nucleus(c_out, c_guess, period, maxsteps, ncpus);
    mpfr_printf("%Re %Re\n", mpc_realref(c_out), mpc_imagref(c_out));
    mpc_clear(c_out);
    mpc_clear(c_guess);
    return 0;
  }
  return 1;
}
```

26 c/bin/m-parent.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

#include <stdio.h>
#include <mandelbrot-numerics.h>
#include "m-util.h"

static void usage(const char *progname) {
  fprintf
    ( stderr
    , "usage: %s precision nucleus-re nucleus-im period maxsteps ncpus\n"
    , progname
    );
}

extern int main(int argc, char **argv) {
  if (argc != 7) {
    usage(argv[0]);
    return 1;
  }
  bool native = true;
  int bits = 0;
  if (! arg_precision(argv[1], &native, &bits)) { return 1; }
  if (native) {
    double cre = 0;
    double cim = 0;
    int period = 0;
    int maxsteps = 0;
    int ncpus = 0;
    if (! arg_double(argv[2], &cre)) { return 1; }
    if (! arg_double(argv[3], &cim)) { return 1; }
    if (! arg_int(argv[4], &period)) { return 1; }
    if (! arg_int(argv[5], &maxsteps)) { return 1; }
    if (! arg_int(argv[6], &ncpus)) { return 1; }
    mpq_t angle;
    mpq_init(angle);
    complex double root = 0;
    complex double parent = 0;
    int p = m_d_parent(angle, &root, &parent, cre + I * cim, period, maxsteps);
    if (p > 0) {
      gmp_printf("%.16e %.16e %Qd %.16e %.16e %d\n", creal(root), cimag(root), ↙
          ↳ angle, creal(parent), cimag(parent), p);
```

```
    } else if (p == 0) {
      printf("%.16e %.16e\n", creal(root), cimag(root));
    }
    mpq_clear(angle);
    return p < 0;
  } else {
    mpc_t c, root, parent;
    mpc_init2(c, bits);
    mpc_init2(root, bits);
    mpc_init2(parent, bits);
    int period = 0;
    int maxsteps = 0;
    int ncpus = 0;
    if (! arg_mpc(argv[2], argv[3], c)) { return 1; }
    if (! arg_int(argv[4], &period)) { return 1; }
    if (! arg_int(argv[5], &maxsteps)) { return 1; }
    if (! arg_int(argv[6], &ncpus)) { return 1; }
    mpq_t angle;
    mpq_init(angle);
    int p = m_r_parent(angle, root, parent, c, period, maxsteps, ncpus);
    if (p > 0) {
      mpfr_printf("%Re %Re %Qd %Re %Re %d\n", mpc_realref(root), mpc_imagref(root), angle, mpc_realref(parent), mpc_imagref(parent), p);
    } else if (p == 0) {
      mpfr_printf("%Re %Re\n", mpc_realref(root), mpc_imagref(root));
    }
    mpc_clear(c);
    mpc_clear(root);
    mpc_clear(parent);
    mpq_clear(angle);
    return p < 0;
  }
  return 1;
}
```

27 c/bin/m-pi-rays.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

#include <stdio.h>
#include <mandelbrot-numerics.h>
#include "m-util.h"

static void usage(const char *progname) {
  fprintf
    ( stderr
    , "usage: %s parent-period nucleus-re nucleus-im internal-angle external-angle\n"
    , progname
    );
}

extern int main(int argc, char **argv) {
  if (argc != 6) {
    usage(argv[0]);
```

```
    return 1;
  }
  int period = 0;
  double nucleusre, nucleusim;
  mpq_t inangle, exangle;
  mpq_init(inangle);
  mpq_init(exangle);
  arg_int(argv[1], &period);
  arg_double(argv[2], &nucleusre);
  arg_double(argv[3], &nucleusim);
  arg_rational(argv[4], inangle);
  arg_rational(argv[5], exangle);
  complex double pnucleus;
  m_d_nucleus(&pnucleus, nucleusre + I * nucleusim, period, 256);
  complex double z, bond;
  m_d_interior(&z, &bond, pnucleus, pnucleus, cexp(I * 2 * 3.141592653589793 * ↙
      ↳ mpq_get_d(inangle)), period, 256);
  complex double cnucleus;
  int den = mpz_get_si(mpq_denref(inangle));
  m_d_nucleus(&cnucleus, bond + (bond - pnucleus) / (den * den) , period * den, ↙
      ↳ 256);
  double s = cabs(cnucleus - bond);
  int sharpness = 4;
  m_d_exray_in *ray = m_d_exray_in_new(exangle, sharpness);
  if (! ray) { mpq_clear(inangle); mpq_clear(exangle); return 1; }
  int n = 0;
  for (int j = 4; j < 17; ++j) {
    for (int i = 0; i < 1 << j; ++i) {
      m_d_exray_in_step(ray, 4);
    }
    complex double c = m_d_exray_in_get(ray);
    double eps = cabs(c - bond);
    z = 0;
    for (n = 0; ; ++n) {
      z = z * z + c;
      if (cabs(z) > 2) { break; }
    }
    printf("%d %.16e %.16e\n", n, eps, eps * n / (2 * s * period * den));
    fflush(stdout);
  }
  m_d_exray_in_delete(ray);
  mpq_clear(inangle);
  mpq_clear(exangle);
  return 0;
}
```

28 c/bin/m-pi-rays.sh

```
#!/bin/bash
m-pi-rays 1  0 0 1/2   1/3   > "1_1-2_1-3.dat"
m-pi-rays 2 -1 0 1/2   2/5   > "2_1-2_2-5.dat"
m-pi-rays 1  0 0 1/3   1/7   > "1_1-3_1-7.dat"
m-pi-rays 1  0 0 1/3   2/7   > "1_1-3_2-7.dat"
m-pi-rays 1  0 0 2/5   9/31  > "1_2-5_9-31.dat"
m-pi-rays 1  0 0 2/5  10/31  > "1_2-5_10-31.dat"
m-pi-rays 3 -2 0 1/2   4/9   > "3_1-2_4-9.dat"
m-pi-rays 3 -2 0 1/3 220/511 > "3_1-3_220-511.dat"
```

29 c/bin/m-rodney.c

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

#include <mandelbrot-numerics.h>

#define U (rand()/(double) RAND_MAX)
#define pi 3.141592653589793
#define twopi (2 * pi)

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

void wheel(const mpc_t c, const mpfr_t r, int p)
{
  mpfr_t zoom;
  mpfr_init2(zoom, 64);
  mpfr_ui_div(zoom, 1, r, MPFR_RNDN);
  mpfr_printf
    ( // location
      "Re: %Re\r\n"
      "Im: %Re\r\n"
      "Zoom: %Re\r\n"
      "Iterations: 10100100\r\n"
      "IterDiv: 0.010000\r\n"
      "SmoothMethod: 0\r\n"
      "ColorMethod: 7\r\n"
      "Differences: 3\r\n"
      "ColorOffset: 0\r\n"
      "Rotate: 0.000000\r\n"
      "Ratio: 360.000000\r\n"
      "Colors: ↙
          ↳ 255,255,255,128,0,64,160,0,0,192,128,0,64,128,0,0,255,255,64,128,255,0,0,255,\↙
          ↳ r\n"
      "InteriorColor: 0,0,0,\r\n"
      "Smooth: 1\r\n"
      "MultiColor: 0\r\n"
      "BlendMC: 0\r\n"
      "MultiColors: \r\n"
      "Power: 2\r\n"
      "FractalType: 0\r\n"
      "Slopes: 1\r\n"
      "SlopePower: 50\r\n"
      "SlopeRatio: 20\r\n"
      "SlopeAngle: 45\r\n"
      "imag: 1\r\n"
      "real: 1\r\n"
      "SeedR: 0\r\n"
      "SeedI: 0\r\n"
```

```
        "FactorAR: 1\r\n"
        "FactorAI: 0\r\n"
        "Period: %d\r\n"
        // settings
        "ZoomSize: 2\r\n"
        "MaxReferences: 10000\r\n"
        "GlitchLowTolerance: 0\r\n"
        "ApproxLowTolerance: 0\r\n"
        "AutoApproxTerms: 1\r\n"
        "ImageWidth: 256\r\n"
        "ImageHeight: 256\r\n"
        "ThreadsPerCore: 1\r\n"
        "ArbitrarySize: 1\r\n"
        "ReuseReference: 0\r\n"
        "AutoSolveGlitches: 1\r\n"
        "Guessing: 1\r\n"
        "SolveGlitchNear: 0\r\n"
        "NoApprox: 0\r\n"
        "Mirror: 0\r\n"
        "LongDoubleAlways: 0\r\n"
        "FloatExpAlways: 0\r\n"
        "AutoIterations: 1\r\n"
        "ShowGlitches: 1\r\n"
        "NoReuseCenter: 1\r\n"
        "IsolatedGlitchNeighbourhood: 4\r\n"
        "JitterSeed: 1\r\n"
        "JitterShape: 0\r\n"
        "JitterScale: 1\r\n"
        "Derivatives: 0\r\n"
        "ShowCrossHair: 0\r\n"
        "UseNanoMB1: 0\r\n"
        "UseNanoMB2: 1\r\n"
        "OrderM: 16\r\n"
        "OrderN: 16\r\n"
        "InteriorChecking: 0\r\n"
        "RadiusScale: 0.1\r\n"
      , mpc_realref(c)
      , mpc_imagref(c)
      , zoom
      , p
      );
  mpfr_clear(zoom);
}

int gcd(int a, int b)
{
  if (a <= 0) return b;
  if (a > b) return gcd(b, a);
  return gcd(b % a, a);
}

int main(int argc, char **argv)
{
  if (argc > 1)
    srand(atoi(argv[1]));
  else
    srand(time(0));
```

```
  int kind = 1 * U;
  switch (kind)
  {
#if 0
    case -1: // shoreline
    {
      double _Complex c = 0;
      int period = 1;
      double size = cabs(m_d_size(c, period));
      do
      {
        int denominator = 2 + 12 * U;
        int numerator = 1 + (denominator - 1) * U;
        int g = gcd(numerator, denominator);
        numerator /= g;
        denominator /= g;
        fprintf(stderr, "q/p = %d/%d\n", numerator, denominator);
        double _Complex z_new = 0;
        double _Complex c_new = 0;
        double _Complex z_guess = 0;
        double _Complex c_guess = c;
        double _Complex interior = cexp(twopi * I * numerator / denominator);
        m_d_interior(&z_new, &c_new, z_guess, c_guess, interior, period, 64);
        c_guess = c_new + (c_new - c_guess) * (period == 1 ? sin(pi * numerator ↙
            ↳ / denominator) : 1) / (denominator * denominator);
        period *= denominator;
        m_d_nucleus(&c, c_guess, period, 64);
        size = cabs(m_d_size(c, period));
      } while (size > 1e-10);
      double _Complex z_new = 0;
      double _Complex c_new = 0;
      double _Complex z_guess = 0;
      double _Complex c_guess = c;
      double _Complex interior = cexp(twopi * I * U);
      m_d_interior(&z_new, &c_new, z_guess, c_guess, interior, period, 64);
      size /= 100;
      //output(c_new, size, period);
      break;
    }
#endif
    default:
    case 0: // wheels
    {
      mpc_t c_guess, c_out;
      mpc_t z_guess, z_out;
      mpc_init2(c_guess, 1024);
      mpc_init2(c_out, 1024);
      mpc_init2(z_guess, 1024);
      mpc_init2(z_out, 1024);
      mpc_t c_outer, c_inner, c_new;
      mpc_init2(c_outer, 1024);
      mpc_init2(c_inner, 1024);
      mpc_init2(c_new, 1024);
      mpc_set_ui(c_outer, 0, MPC_RNDNN);
      mpc_set_dc(c_inner, -2, MPC_RNDNN);
      int outer_period = 1, inner_period = 3 + 10 * U;
      m_r_nucleus(c_inner, c_inner, inner_period, 64, 1);
```

```
        //mpfr_fprintf(stderr, "%Re + %Re i\n", mpc_realref(c_inner), mpc_imagref(↙
            ↳ c_inner));
        mpc_t size, inner_size;
        mpc_init2(size, 64);
        mpc_init2(inner_size, 64);
        mpfr_t outer_r, inner_r, new_r;
        mpfr_init2(outer_r, 64);
        mpfr_init2(inner_r, 64);
        mpfr_init2(new_r, 64);
        m_r_size(size, c_outer, outer_period);
        mpc_abs(outer_r, size, MPFR_RNDN);
        m_r_size(inner_size, c_inner, inner_period);
        mpc_abs(inner_r, inner_size, MPFR_RNDN);
        mpfr_t o, i;
        mpfr_init2(o, 64);
        mpfr_init2(i, 64);
        mpc_t t;
        mpc_init2(t, 64);
        int depth;
        for (depth = 0; depth < 16; ++depth)
        {
          mpfr_prec_t p = 53 - 2 * mpfr_get_exp(inner_r);
          mpfr_prec_round(mpc_realref(c_inner), p, MPFR_RNDN);
          mpfr_prec_round(mpc_imagref(c_inner), p, MPFR_RNDN);
          mpfr_set_prec(mpc_realref(c_guess), p);
          mpfr_set_prec(mpc_imagref(c_guess), p);
          mpfr_set_prec(mpc_realref(c_out), p);
          mpfr_set_prec(mpc_imagref(c_out), p);
          mpfr_set_prec(mpc_realref(z_guess), p);
          mpfr_set_prec(mpc_imagref(z_guess), p);
          mpfr_set_prec(mpc_realref(z_out), p);
          mpfr_set_prec(mpc_imagref(z_out), p);
          mpfr_set_prec(mpc_realref(c_new), 3 * p);
          mpfr_set_prec(mpc_imagref(c_new), 3 * p);
          mpfr_pow_ui(i, inner_r, 3, MPFR_RNDN);
          mpfr_fprintf(stderr, "%d @ %Re\n", inner_period, inner_r);
          if (inner_period >= 10000)
          {
            break;
          }
          for (int try = 0; try < 10000; ++try)
          {
            // pick a point outside M_1
            double _Complex c = (4 * U - 2) + I * (4 * U - 2) - 0.75;
            double _Complex z = 0;
            int k;
            for (k = 0; k < 100; ++k)
            {
              z = z * z + c;
              if (cnorm(z) > 4) break;
            }
            if (! (cnorm(z) > 4)) continue;
            if (k < 10) continue;
#if 0
            // compute atom coordinate
            z = 0;
            for (k = 0; k < 64; ++k)
```

```
            m_d_attractor(&z, z, (k + 0.5)/64 * c, 1, 64);
          m_d_attractor(&z, z, c, 1, 64);
          double _Complex a = 2 * z;
#endif
          //fprintf(stderr, "%e + %e i\n", creal(a), cimag(a));
          // compute coordinate relative to inner atom
          mpc_set_dc(t, c, MPC_RNDNN);
          mpc_mul(t, t, inner_size, MPC_RNDNN);
          mpc_add(c_out, c_inner, t, MPC_RNDNN);
#if 0
          mpc_set_dc(z_guess, 0, MPC_RNDNN);
          mpc_set(c_guess, c_inner, MPC_RNDNN);
          for (int k = 0; k < 16; ++k)
          {
            mpc_set_dc(t, (k + 0.5)/16 * a, MPC_RNDNN);
            m_r_interior(z_out, c_out, z_guess, c_guess, t, inner_period, 64);
            mpc_swap(z_out, z_guess);
            mpc_swap(c_out, c_guess);
          }
          mpc_set_dc(t, a, MPC_RNDNN);
          m_r_interior(z_out, c_out, z_guess, c_guess, t, inner_period, 64);
#endif
          // compute period
          mpfr_mul_d(new_r, inner_r, 1e-4, MPFR_RNDN);
          int new_period = m_r_ball_period_do(c_out, new_r, inner_period * 100);
          if (new_period <= inner_period)
            continue;
          if (new_period % inner_period == 0)
            continue;
          // compute nucleus
          m_newton result = m_r_nucleus(c_new, c_out, new_period, 256, 1);
          if (result == m_failed)
            continue;
          m_r_size(size, c_new, new_period);
          mpc_abs(new_r, size, MPFR_RNDN);
          if (! mpfr_greater_p(new_r, i))
            continue;
          if (! mpfr_less_p(new_r, inner_r))
            continue;
          // step
          mpc_swap(inner_size, size);
          mpc_swap(c_inner, c_outer);
          mpc_swap(c_inner, c_new);
          outer_period = inner_period;
          inner_period = new_period;
          mpfr_swap(inner_r, outer_r);
          mpfr_swap(inner_r, new_r);
          break;
        }
      }
      mpfr_prec_t p = 53 - 2 * mpfr_get_exp(inner_r);
      mpfr_prec_round(mpc_realref(c_inner), p, MPFR_RNDN);
      mpfr_prec_round(mpc_imagref(c_inner), p, MPFR_RNDN);
      if (depth == 16)
        exit(1);
      int fold = 7 * U;
      switch (fold)
```

```
280         {
              case 0: // wheel
                fprintf(stderr, "wheel\n");
                mpfr_mul_d(new_r, inner_r, 100, MPFR_RNDN);
                break;
285           case 1: // 2-fold
              case 2: // 4-fold
              case 3: // 8-fold
                fprintf(stderr, "%d-fold\n", 1 << fold);
                //int new_period = outer_period + fold * inner_period;
290             mpfr_set(new_r, outer_r, MPFR_RNDN);
                for (int k = 0; k < fold; ++k)
                {
                  mpfr_mul(new_r, new_r, inner_r, MPFR_RNDN);
                  mpfr_sqrt(new_r, new_r, MPFR_RNDN);
295             }
    #if 0
                for (int k = 0; k < 100; ++k)
                {
                  double _Complex a = cexp(twopi * I * U);
300               mpc_set_dc(t, a, MPC_RNDNN);
                  mpc_mul_fr(t, t, new_r, MPC_RNDNN);
                  mpc_add(c_guess, c_inner, t, MPC_RNDNN);
                  m_newton result = m_r_nucleus(c_new, c_guess, new_period, 64, 1);
                  if (result == m_failed)
305                 continue;
                  mpc_sub(t, c_new, c_inner, MPC_RNDNN);
                  mpc_abs(o, t, MPFR_RNDN);
                  if (mpfr_greater_p(inner_r, o))
                    continue;
310               if (mpfr_greater_p(o, outer_r))
                    continue;
                  mpfr_swap(o, new_r);
                  break;
                }
315 #endif
    //            mpfr_mul_2ui(new_r, new_r, 1, MPFR_RNDN);
                break;
              case 4: // minibrot
                fprintf(stderr, "minibrot\n");
320             mpc_set_dc(z_out, 0, MPC_RNDNN);
                mpc_set_dc(t, -1, MPC_RNDNN);
                m_r_interior(z_out, c_inner, z_out, c_inner, t, inner_period, 64);
                mpfr_mul_d(new_r, inner_r, 1, MPFR_RNDN);
                break;
325           case 5: // shoreline
              case 6: // texture
                fprintf(stderr, fold == 5 ? "shoreline\n" : "texture\n");
                // pick a point outside M_1
                double _Complex c, z;
330             for (int j = 0; j < 10000; ++j)
                {
                  c = (4 * U - 2) + I * (4 * U - 2) - 0.75;
                  z = 0;
                  int k;
335               for (k = 0; k < 100; ++k)
                  {
```

```
            z = z * z + c;
            if (cnorm(z) > 4) break;
          }
          if (! (cnorm(z) > 4)) continue;
          if (k < 10) continue;
        }
        // compute atom coordinate
#if 0
        z = 0;
        for (int k = 0; k < 64; ++k)
          m_d_attractor(&z, z, (k + 0.5)/64 * c, 1, 64);
        m_d_attractor(&z, z, c, 1, 64);
        double _Complex a = 2 * z;
#endif
        //fprintf(stderr, "%e + %e i\n", creal(a), cimag(a));
        // compute coordinate relative to inner atom
        mpc_set_dc(t, c, MPC_RNDNN);
        mpc_mul(t, t, inner_size, MPC_RNDNN);
        mpc_add(c_inner, c_inner, t, MPC_RNDNN);
        mpfr_mul_d(new_r, inner_r, fold == 5 ? 0.1 : 0.01, MPFR_RNDN);
#if 0
        int new_period = inner_period;
        int count = 4 * U;
        for (int k = 0; k < count; ++k)
        {
          int denominator = 2 + 11 * U;
          int numerator = 1 + (denominator - 1) * U;
          int g = gcd(numerator, denominator);
          numerator /= g;
          denominator /= g;
          fprintf(stderr, "q/p = %d/%d\n", numerator, denominator);
          mpc_set_dc(t, cexp(twopi * I * numerator / denominator), MPC_RNDNN);
          mpc_set_dc(z_out, 0, MPC_RNDNN);
          m_r_interior(z_out, c_new, z_out, c_inner, t, new_period, 64);
          mpc_sub(t, c_new, c_inner, MPC_RNDNN);
          mpfr_set_d(i, (k == 0 ? sin(pi * numerator / denominator) : 1) / (↙
              ↳ denominator * denominator), MPFR_RNDN);
          mpc_mul_fr(t, t, i, MPC_RNDNN);
          mpc_add(c_inner, c_new, t, MPC_RNDNN);
          new_period *= denominator;
        }
        m_r_nucleus(c_inner, c_inner, new_period, 64, 1);
        m_r_size(size, c_inner, new_period);
        mpc_abs(new_r, size, MPFR_RNDN);
        mpfr_mul_d(new_r, new_r, 4, MPFR_RNDN);
#endif
        break;
      }
      wheel(c_inner, new_r, inner_period);
      break;
    }
  }
  return 0;
}
```

30 c/bin/m-shape.c

```c
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

#include <stdio.h>
#include <mandelbrot-numerics.h>
#include "m-util.h"

static void usage(const char *progname) {
  fprintf
    ( stderr
    , "usage: %s precision nucleus-re nucleus-im period\n"
    , progname
    );
}

extern int main(int argc, char **argv) {
  if (argc != 5) {
    usage(argv[0]);
    return 1;
  }
  bool native = true;
  int bits = 0;
  if (! arg_precision(argv[1], &native, &bits)) { return 1; }
  if (native) {
    double nre = 0;
    double nim = 0;
    int period = 0;
    if (! arg_double(argv[2], &nre)) { return 1; }
    if (! arg_double(argv[3], &nim)) { return 1; }
    if (! arg_int(argv[4], &period)) { return 1; }
    double _Complex s = m_d_shape_estimate(nre + I * nim, period);
    m_shape shape = m_d_shape_discriminant(s);
    switch (shape) {
      case m_cardioid: printf("cardioid %.16e %.16e\n", creal(s), cimag(s)); ↙
          ↳ return 0;
      case m_circle:   printf("circle %.16e %.16e\n",   creal(s), cimag(s)); ↙
          ↳ return 0;
    }
  } else {
    mpc_t n;
    int period = 0;
    mpc_init2(n, bits);
    if (! arg_mpc(argv[2], argv[3], n)) { return 1; }
    if (! arg_int(argv[4], &period)) { return 1; }
    mpc_t s;
    mpc_init2(s, 53);
    m_r_shape_estimate(s, n, period);
    m_shape shape = m_r_shape_discriminant(s);
    mpc_clear(n);
    switch (shape) {
      case m_cardioid: mpfr_printf("cardioid %Re %Re\n", mpc_realref(s), ↙
          ↳ mpc_imagref(s)); break;
      case m_circle:   mpfr_printf("circle %Re %Re\n"  , mpc_realref(s), ↙
          ↳ mpc_imagref(s)); break;
    }
    mpc_clear(s);
```

```
  }
  return 1;
}
```

31 c/bin/m-size.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

#include <stdio.h>
#include <mandelbrot-numerics.h>
#include "m-util.h"

static void usage(const char *progname) {
  fprintf
    ( stderr
    , "usage:\n"
      "%s precision nucleus-re nucleus-im period\n"
      "%s precision period <<EOF\n"
      "nucleus-re\nnucleus-im\nEOF\n"
    , progname, progname
    );
}

extern int main(int argc, char **argv) {
  if (argc != 3 && argc != 5) {
    usage(argv[0]);
    return 1;
  }
  bool native = true;
  int bits = 0;
  if (! arg_precision(argv[1], &native, &bits)) { return 1; }
  if (native) {
    double nre = 0;
    double nim = 0;
    int period = 0;
    if (argc == 3)
    {
      if (! arg_int(argv[2], &period)) { return 1; }
      if (! stdin_double(&nre)) { return 1; }
      if (! stdin_double(&nim)) { return 1; }
    }
    else
    {
      if (! arg_double(argv[2], &nre)) { return 1; }
      if (! arg_double(argv[3], &nim)) { return 1; }
      if (! arg_int(argv[4], &period)) { return 1; }
    }
    complex double size = m_d_size(nre + I * nim, period);
    printf("%.16e %.16e\n", cabs(size), carg(size));
    return 0;
  } else {
    mpc_t n;
    int period = 0;
    mpc_init2(n, bits);
    if (argc == 3)
```

```
    {
      if (! arg_int(argv[2], &period)) { return 1; }
      if (! stdin_mpfr(mpc_realref(n))) { return 1; }
      if (! stdin_mpfr(mpc_imagref(n))) { return 1; }
    }
    else
    {
      if (! arg_mpc(argv[2], argv[3], n)) { return 1; }
      if (! arg_int(argv[4], &period)) { return 1; }
    }
    mpc_t size;
    mpc_init2(size, 53);
    m_r_size(size, n, period);
    mpfr_t r, t;
    mpfr_init2(r, 53);
    mpfr_init2(t, 53);
    mpc_abs(r, size, MPFR_RNDN);
    mpc_arg(t, size, MPFR_RNDN);
    mpfr_printf("%Re %Re\n", r, t);
    mpfr_clear(r);
    mpfr_clear(t);
    mpc_clear(size);
    mpc_clear(n);
    return 0;
  }
  return 1;
}
```

32 c/bin/m-size-precision.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

/*

for antenna:
need nucleus precision at least 2x period to compute size and domain size
need nucleus precision at least 4x period to ensure nucleus within atom
atom size ~ A/16^period
domain size ~ B/4^period ~ sqrt(atom size)

*/
#include <stdio.h>
#include <mandelbrot-numerics.h>

int main(int argc, char **argv)
{
  (void) argc;
  (void) argv;
  for (int period = 1; period < 64; period += 1)
  {
    printf("# period %d\n", period);
    fflush(stdout);
    mpfr_prec_t prec = 16 * period + 8;
    mpc_t guess, nucleus, size;
    mpc_init2(guess, prec);
```

```
    mpc_init2(nucleus, prec);
    mpc_init2(size, prec);
    mpc_set_d(guess, -2, MPC_RNDNN);
    while (prec > 2)
    {
      mpfr_prec_round(mpc_realref(guess), prec, MPFR_RNDN);
      mpfr_prec_round(mpc_imagref(guess), prec, MPFR_RNDN);
      m_r_nucleus(nucleus, guess, period, 64, 8);
      printf("%ld ", prec);
      fflush(stdout);
      m_r_size(size, nucleus, period);
      mpc_norm(mpc_realref(size), size, MPFR_RNDN);
      mpfr_log2(mpc_realref(size), mpc_realref(size), MPFR_RNDN);
      mpfr_div_2ui(mpc_realref(size), mpc_realref(size), 1, MPFR_RNDN);
      mpfr_printf("%Re ", mpc_realref(size));
      fflush(stdout);
      m_r_domain_size(mpc_realref(size), nucleus, period);
      mpfr_prec_round(mpc_realref(size), 53, MPFR_RNDN);
      mpfr_log2(mpc_realref(size), mpc_realref(size), MPFR_RNDN);
      mpfr_printf("%Re\n", mpc_realref(size));
      fflush(stdout);
      prec--;
      mpfr_prec_round(mpc_realref(nucleus), prec, MPFR_RNDN);
      mpfr_prec_round(mpc_imagref(nucleus), prec, MPFR_RNDN);
    }
    printf("\n\n");
    fflush(stdout);
  }
  return 0;
}
```

33 c/bin/m-sonify.c

```
#include <mandelbrot-numerics.h>
#include <math.h>
#include <sndfile.h>

#define SR 48000
#define FPS 60
#define PI 3.141592653589793

// http://burtleburtle.net/bob/hash/integer.html
static uint32_t burtle_hash(uint32_t a)
{
    a = (a+0x7ed55d16) + (a<<12);
    a = (a^0xc761c23c) ^ (a>>19);
    a = (a+0x165667b1) + (a<<5);
    a = (a+0xd3a2646c) ^ (a<<9);
    a = (a+0xfd7046c5) + (a<<3);
    a = (a^0xb55a4f09) ^ (a>>16);
    return a;
}

// pseudo-random uniform number in [0,1)
static float uniform(uint32_t c)
{
//  return rand() / (double) RAND_MAX;
```

```
  return burtle_hash(c) / (float) (0x100000000LL);
}

float buffer[SR][4];

int main(int argc, char **argv)
{
  SF_INFO info = { 0, 48000, 4, SF_FORMAT_WAV | SF_FORMAT_FLOAT, 0, 0 };
  SNDFILE *sndfile = sf_open("m-sonify.wav", SFM_WRITE, &info);
  mpq_t angle;
  mpq_init(angle);
  int sharpness = 8;
  int j = 0;
  for (int period = 1; period < 8; ++period)
  {
    int denominator = (1 << period) - 1;
    for (int numerator = 1; numerator < denominator; numerator += 2)
    {
      mpq_set_ui(angle, numerator, denominator);
      mpq_canonicalize(angle);
      if (mpz_get_ui(mpq_denref(angle)) != denominator) continue;
      m_d_exray_in *ray = m_d_exray_in_new(angle, sharpness);
      for (int i = 0; i < period * sharpness * 2; ++i)
        m_d_exray_in_step(ray, 16);
      double _Complex endpoint = m_d_exray_in_get(ray);
      m_d_exray_in_delete(ray);
      double _Complex nucleus;
      m_d_nucleus(&nucleus, endpoint, period, 16);
      int k = 0;
      double _Complex z = 0;
      double _Complex c = nucleus;
      for (int t = 0; t < SR; ++t)
      {
        m_d_interior(&z, &c, z, c, cexp(I * 2 * PI * (t + 0.5) / SR), period, ↙
            ↳ 16);

  buffer[k][0] = 0;
  buffer[k][1] = 0;
  buffer[k][2] = 0;
  buffer[k][3] = 0;
  float omni = (uniform(j++) - 0.5f) / period;
        for (int i = 0; i < period; ++i)
        {
  double X = creal(z);
  double Y = cimag(z);
  double R = X * X + Y * Y + 1;
  double u = 2 * X / R;
  double v = 2 * Y / R;
  double w = (R - 2) / R;
  float left  = -u * omni;
  float front = -w * omni;
  float up    =  v * omni;
  buffer[k][0] += omni;
  buffer[k][1] += left;
  buffer[k][2] += up;
  buffer[k][3] += front;
          z = z * z + c;
```

```
        }
          if (++k == SR)
          {
            k = 0;
            sf_writef_float(sndfile, &buffer[0][0], SR);
          }

      }
    }
  }
  mpq_clear(angle);
  sf_close(sndfile);
  return 0;
}
```

34 c/bin/m-stability.c

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

#include <mpfr.h>

int main(int argc, char **argv)
{
  uint64_t exact = 0;
  uint64_t histogram[256];
  for (uint64_t k = 0; k < 256; ++k)
  {
    histogram[k] = 0;
  }
  uint64_t otherwise = 0;
  uint64_t total = 0;
  const uint64_t depth = 11;
  const mpfr_prec_t prec = depth + 1;
  {
    const uint64_t range = 1 << depth;
    #pragma omp parallel for
    for (uint64_t j = 0; j < range; ++j)
    {
      mpfr_t fx, fy, fp, fm, fr, fi;
      mpfr_init2(fx, prec);
      mpfr_init2(fy, prec);
      mpfr_init2(fp, prec);
      mpfr_init2(fm, prec);
      mpfr_init2(fr, prec);
      mpfr_init2(fi, prec);
      mpfr_set_d(fy, (((j << 1) + 1.) / (range << 1)) * 4. - 2., MPFR_RNDN);
      const float dy = mpfr_get_d(fy, MPFR_RNDN);
      for (uint64_t i = 0; i < range; ++i)
      {
        mpfr_set_d(fx, (((i << 1) + 1.) / (range << 1)) * 4. - 2., MPFR_RNDN);
        const double dx = mpfr_get_d(fx, MPFR_RNDN);

        mpfr_add(fp, fx, fy, MPFR_RNDN);
        mpfr_add(fm, fx, fy, MPFR_RNDN);
        mpfr_mul(fr, fp, fm, MPFR_RNDN);
```

```
          mpfr_mul(fi, fx, fy, MPFR_RNDN);
          mpfr_mul_2exp(fi, fi, 1, MPFR_RNDN);

          const double ex = (dx + dy) * (dx - dy);
          const double ey = 2. * dx * dy;

          const double hx = mpfr_get_d(fr, MPFR_RNDN);
          const double hy = mpfr_get_d(fi, MPFR_RNDN);

          const double cx = hx - ex;
          const double cy = hy - ey;
          const double c = cx * cx + cy * cy;

          int b = 0;
          frexp(c, &b);
          const int k = -b;
          if (c == 0.0)
          {
            #pragma omp atomic
            exact++;
          }
          else if (0 <= k && k < 256)
          {
            #pragma omp atomic
            histogram[k]++;
          }
          else
          {
            #pragma omp atomic
            otherwise++;
          }
        }
        mpfr_clear(fx);
        mpfr_clear(fy);
        mpfr_clear(fp);
        mpfr_clear(fm);
        mpfr_clear(fr);
        mpfr_clear(fi);
      }
      total += range * range;
      for (uint64_t k = 0; k < 256; ++k)
      {
        printf("%g\n", histogram[k] / (double) total);
      }
      printf("\n\n");
      fflush(stdout);
    }
    return 0;
}
```

35 c/bin/m-two-spirals-out.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

#include <stdio.h>
```

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

int main()
{
  int n = 64;
  double _Complex c = -1.7687402565964134e+00 + I * 3.0387357058642636e-03;
  int p = 3;
  int P = 56;
  int arms = 2;
  // period 56 nucleus at center
  m_d_nucleus(&c, c, P, n);
  // a
  m_d_misiurewicz(&c, c, P + 1, p, n);
  // b
  m_d_nucleus(&c, c, P = P + arms * p, n);
  // c
  m_d_misiurewicz(&c, c, P + 1, p, n);
  // d
  m_d_nucleus(&c, c, P = P + p, n);
  // e
  m_d_misiurewicz(&c, c, P + 1, p, n);
  // f
  m_d_nucleus(&c, c, P = P + p, n);
  // period 68 nucleus two spirals out
  printf("%.18f %.18f 1e-8\n", creal(c), cimag(c));
  return 0;
}
```

36 c/bin/m-util.h

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

#ifndef M_UTIL_H
#define M_UTIL_H 1

#include <errno.h>
#include <stdbool.h>
#include <stdlib.h>
#include <string.h>

#include <mpfr.h>

static inline bool arg_precision(const char *arg, bool *native, int *bits) {
  if (0 == strcmp("double", arg)) {
    *native = true;
    *bits = 53;
    return true;
  } else {
    char *check = 0;
    errno = 0;
    long int li = strtol(arg, &check, 10);
    bool valid = ! errno && arg != check && ! *check;
    int i = li;
    if (valid && i > 1) {
      *native = false;
```

```
      *bits = i;
      return true;
    }
  }
  return false;
}

static inline bool arg_double(const char *arg, double *x) {
  char *check = 0;
  errno = 0;
  double d = strtod(arg, &check);
  if (! errno && arg != check && ! *check) {
    *x = d;
    return true;
  }
  return false;
}

static inline bool arg_int(const char *arg, int *x) {
  char *check = 0;
  errno = 0;
  long int li = strtol(arg, &check, 10);
  if (! errno && arg != check && ! *check) {
    *x = li;
    return true;
  }
  return false;
}

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

static inline bool arg_mpfr(const char *arg, mpfr_t x) {
  return 0 == mpfr_set_str(x, arg, 10, MPFR_RNDN);
}

static inline bool arg_mpc(const char *re, const char *im, mpc_t x) {
  int ok
    = mpfr_set_str(mpc_realref(x), re, 10, MPFR_RNDN)
    + mpfr_set_str(mpc_imagref(x), im, 10, MPFR_RNDN);
  return ok == 0;
}

static inline bool stdin_double(double *x) {
  return 1 == scanf("%lf", x);
}

static inline bool stdin_mpfr(mpfr_t x) {
  return 0 != mpfr_inp_str(x, stdin, 10, MPFR_RNDN);
}

static const double twopi = 6.283185307179586;

#endif
```

37 c/include/mandelbrot-numerics.h

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2019 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

#ifndef MANDELBROT_NUMERICS_H
#define MANDELBROT_NUMERICS_H 1

#include <complex.h>
#include <math.h>
#include <stdbool.h>
#include <stdint.h>
#include <stdlib.h>
#include <gmp.h>
#include <mpfr.h>
#include <mpc.h>

#ifdef __cplusplus
extern "C"
{
#endif

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

struct m_d_mat2 {
  double _Complex a, b, c, d;
};
typedef struct m_d_mat2 m_d_mat2;

extern void m_d_mat2_set(m_d_mat2 *o, const m_d_mat2 *m);
extern void m_d_mat2_id(m_d_mat2 *o);
extern double _Complex m_d_mat2_tr(const m_d_mat2 *m);
extern double _Complex m_d_mat2_det(const m_d_mat2 *m);
extern void m_d_mat2_inv(m_d_mat2 *m1, const m_d_mat2 *m);
extern void m_d_mat2_mul(m_d_mat2 *o, const m_d_mat2 *l, const m_d_mat2 *r);
extern void m_d_mat2_diagonalize(m_d_mat2 *p, m_d_mat2 *d, m_d_mat2 *p1, const m_d_mat2 *m);
extern void m_d_mat2_moebius3(m_d_mat2 *m, double _Complex zero, double _Complex one, double _Complex infinity);

struct m_d_mat2_interp;
typedef struct m_d_mat2_interp m_d_mat2_interp;

extern m_d_mat2_interp *m_d_mat2_interp_new(void);
extern void m_d_mat2_interp_delete(m_d_mat2_interp *i);
extern void m_d_mat2_interp_init(m_d_mat2_interp *i, const m_d_mat2 *f, const m_d_mat2 *g);
extern void m_d_mat2_interp_do(m_d_mat2 *m, const m_d_mat2_interp *i, double t);

enum m_shape { m_cardioid, m_circle };
typedef enum m_shape m_shape;

enum m_newton { m_failed, m_stepped, m_converged };
typedef enum m_newton m_newton;

struct m_period_filter_t;
```

```
typedef struct m_period_filter_t m_period_filter_t;
typedef bool (m_period_filter_f)(m_period_filter_t *filter, int period);
struct m_period_filter_t
{
  m_period_filter_f *accept;
  m_period_filter_f *reject;
};

/* double precision: m_d_*()  */
/* functions taking non-const mpq_t use them for output */
/* functions taking pointers to double _Complex use them for output */

extern m_newton m_d_nucleus_naive_step(double _Complex *c_out, double _Complex c_guess, int period);
extern m_newton m_d_nucleus_naive(double _Complex *c_out, double _Complex c_guess, int period, int maxsteps);

extern m_newton m_d_nucleus_step(double _Complex *c_out, double _Complex c_guess, int period);
extern m_newton m_d_nucleus(double _Complex *c_out, double _Complex c_guess, int period, int maxsteps);

extern m_newton m_d_misiurewicz_naive_step(double _Complex *c_out, double _Complex c_guess, int preperiod, int period);
extern m_newton m_d_misiurewicz_naive(double _Complex *c_out, double _Complex c_guess, int preperiod, int period, int maxsteps);

extern m_newton m_d_misiurewicz_step(double _Complex *c_out, double _Complex c_guess, int preperiod, int period);
extern m_newton m_d_misiurewicz(double _Complex *c_out, double _Complex c_guess, int preperiod, int period, int maxsteps);

extern m_newton m_d_attractor_step(double _Complex *z, double _Complex z_guess, double _Complex c, int period);
extern m_newton m_d_attractor(double _Complex *z, double _Complex z_guess, double _Complex c, int period, int maxsteps);

extern m_newton m_d_interior_step(double _Complex *z, double _Complex *c, double _Complex z_guess, double _Complex c_guess, double _Complex interior, int period);
extern m_newton m_d_interior(double _Complex *z, double _Complex *c, double _Complex z_guess, double _Complex c_guess, double _Complex interior, int period, int maxsteps);

extern m_newton m_d_domain_coord_step(double _Complex *c_out, double _Complex c_guess, double _Complex domain_coord, int loperiod, int hiperiod);
extern m_newton m_d_domain_coord(double _Complex *c_out, double _Complex c_guess, double _Complex domain_coord, int loperiod, int hiperiod, int maxsteps);

extern int m_d_parent(mpq_t angle, double _Complex *root_out, double _Complex *parent_out, double _Complex nucleus, int period, int maxsteps);

extern bool m_d_interior_de(double *de_out, double _Complex *dz_out, double _Complex z, double _Complex c, int p, int steps);

extern double _Complex m_d_size(double _Complex nucleus, int period);
extern double m_d_domain_size(double _Complex nucleus, int period);
```

```
extern double m_d_filtered_domain_size(double _Complex nucleus, int period, ↙
    ↳ m_period_filter_t *filter);
extern m_shape m_d_shape(double _Complex nucleus, int period);
extern double _Complex m_d_shape_estimate(double _Complex nucleus, int period);
extern m_shape m_d_shape_discriminant(double _Complex shape);

struct m_d_exray_in;
typedef struct m_d_exray_in m_d_exray_in;
extern m_d_exray_in *m_d_exray_in_new(const mpq_t angle, int sharpness);
extern void m_d_exray_in_delete(m_d_exray_in *ray);
extern m_newton m_d_exray_in_step(m_d_exray_in *ray, int maxsteps);
extern double _Complex m_d_exray_in_get(const m_d_exray_in *ray);
extern double _Complex m_d_exray_in_do(const mpq_t angle, int sharpness, int ↙
    ↳ maxsteps, int maxnewtonsteps);

struct m_d_exray_out;
typedef struct m_d_exray_out m_d_exray_out;
extern m_d_exray_out *m_d_exray_out_new(double _Complex c, int sharpness, int ↙
    ↳ maxdwell);
extern void m_d_exray_out_delete(m_d_exray_out *ray);
extern m_newton m_d_exray_out_step(m_d_exray_out *ray);
extern bool m_d_exray_out_have_bit(const m_d_exray_out *ray);
extern bool m_d_exray_out_get_bit(const m_d_exray_out *ray);
extern double _Complex m_d_exray_out_get(const m_d_exray_out *ray);
extern char *m_d_exray_out_do(double _Complex c, int sharpness, int maxdwell);

struct m_d_equipotential;
typedef struct m_d_equipotential m_d_equipotential;
extern m_d_equipotential *m_d_equipotential_new(const double _Complex c, int ↙
    ↳ sharpness, int count);
extern void m_d_equipotential_delete(m_d_equipotential *ray);
extern m_newton m_d_equipotential_step(m_d_equipotential *ray, int maxsteps);
extern double _Complex m_d_equipotential_get(const m_d_equipotential *ray);

struct m_d_box_period;
typedef struct m_d_box_period m_d_box_period;
extern m_d_box_period *m_d_box_period_new(double _Complex center, double radius)↙
    ↳ ;
extern void m_d_box_period_delete(m_d_box_period *box);
extern bool m_d_box_period_step(m_d_box_period *box);
extern bool m_d_box_period_have_period(const m_d_box_period *box);
extern int m_d_box_period_get_period(const m_d_box_period *box);
extern int m_d_box_period_do(double _Complex center, double radius, int ↙
    ↳ maxperiod);

struct m_d_box_misiurewicz;
typedef struct m_d_box_misiurewicz m_d_box_misiurewicz;
extern m_d_box_misiurewicz *m_d_box_misiurewicz_new(double _Complex center, ↙
    ↳ double radius);
extern void m_d_box_misiurewicz_delete(m_d_box_misiurewicz *box);
extern bool m_d_box_misiurewicz_step(m_d_box_misiurewicz *box);
extern bool m_d_box_misiurewicz_check_periods(m_d_box_misiurewicz *box, int ↙
    ↳ maxperiod);
extern int m_d_box_misiurewicz_get_preperiod(const m_d_box_misiurewicz *box);
extern int m_d_box_misiurewicz_get_period(const m_d_box_misiurewicz *box);
extern bool m_d_box_misiurewicz_do(int *preperiod, int *period, double _Complex ↙
    ↳ center, double radius, int maxpreperiod, int maxperiod);
```

```
140
struct m_d_ball_period;
typedef struct m_d_ball_period m_d_ball_period;
extern m_d_ball_period *m_d_ball_period_new(double _Complex center, double ↙
    ↳ radius);
extern void m_d_ball_period_delete(m_d_ball_period *ball);
145 extern bool m_d_ball_period_step(m_d_ball_period *ball);
extern bool m_d_ball_period_have_period(const m_d_ball_period *ball);
extern int m_d_ball_period_get_period(const m_d_ball_period *ball);
extern int m_d_ball_period_do(double _Complex center, double radius, int ↙
    ↳ maxperiod);

150 extern void m_d_external_angles(char **lo, char **hi, double _Complex nucleus, ↙
    ↳ int period, double radius, int resolution);

extern double m_d_from_logistic(double r);
extern double m_d_to_logistic(double c);

155 /* arbitrary precision: m_r_*() */
/* functions taking non-const mpq_t/mpfr_t/mpc_t use them for output, _raw also ↙
    ↳ uses for temporaries */

extern m_newton m_r_nucleus_naive_step_raw(mpc_t c_out, const mpc_t c_guess, int↙
    ↳  period, mpc_t z, mpc_t dc, mpc_t c_new, mpc_t d, mpfr_t d2, mpfr_t ↙
    ↳ epsilon2);
extern m_newton m_r_nucleus_naive_step(mpc_t c_out, const mpc_t c_guess, int ↙
    ↳ period);
160 extern m_newton m_r_nucleus_naive(mpc_t c_out, const mpc_t c_guess, int period, ↙
    ↳ int maxsteps, int ncpus);

extern m_newton m_r_nucleus_step_raw(mpc_t c_out, const mpc_t c_guess, int ↙
    ↳ period, mpfr_t epsilon2, mpc_t z, mpc_t dc, mpc_t c_new, mpc_t h, mpc_t dh↙
    ↳ , mpc_t f, mpc_t df);
//extern m_newton m_r_nucleus_step(mpc_t c_out, const mpc_t c_guess, int period)↙
    ↳ ;
extern m_newton m_r_nucleus(mpc_t c_out, const mpc_t c_guess, int period, int ↙
    ↳ maxsteps, int ncpus);
165
extern m_newton m_r_misiurewicz_naive_step_raw(mpc_t c_out, const mpc_t c_guess,↙
    ↳  int preperiod, int period, mpc_t z, mpc_t dc, mpc_t zp, mpc_t dcp, mpc_t ↙
    ↳ c_new, mpc_t d, mpfr_t d2, mpfr_t epsilon2);
extern m_newton m_r_misiurewicz_naive_step(mpc_t c_out, const mpc_t c_guess, int↙
    ↳  preperiod, int period);
extern m_newton m_r_misiurewicz_naive(mpc_t c_out, const mpc_t c_guess, int ↙
    ↳ preperiod, int period, int maxsteps);
extern m_newton m_r_misiurewicz_step_raw(mpc_t c_out, const mpc_t c_guess, int ↙
    ↳ preperiod, int period, mpc_t z, mpc_t dc, mpc_t zp, mpc_t dcp, mpc_t f, ↙
    ↳ mpc_t df, mpc_t g, mpc_t dg, mpc_t h, mpc_t dh, mpc_t k, mpc_t l, mpfr_t ↙
    ↳ epsilon2);
170 // extern m_newton m_r_misiurewicz_step(mpc_t c_out, const mpc_t c_guess, int ↙
    ↳ preperiod, int period, int maxsteps);
extern m_newton m_r_misiurewicz(mpc_t c_out, const mpc_t c_guess, int preperiod,↙
    ↳  int period, int maxsteps);

extern m_newton m_r_attractor_step_raw(mpc_t z_out, const mpc_t z_guess, const ↙
    ↳ mpc_t c, int period, mpc_t z, mpc_t dz, mpc_t z_new, mpc_t d, mpfr_t d2, ↙
    ↳ mpfr_t epsilon2);
```

```
extern m_newton m_r_attractor_step(mpc_t z_out, const mpc_t z_guess, const mpc_t c, int period);
extern m_newton m_r_attractor(mpc_t z_out, const mpc_t z_guess, const mpc_t c, int period, int maxsteps);

extern m_newton m_r_interior_step_raw(mpc_t z_out, mpc_t c_out, const mpc_t z_guess, const mpc_t c_guess, const mpc_t interior, int period, mpc_t z_new, mpc_t c_new, mpc_t c, mpc_t z, mpc_t dz, mpc_t dc, mpc_t dzdz, mpc_t dcdz, mpc_t det, mpc_t d, mpc_t dz1, mpfr_t d2z, mpfr_t d2c, mpfr_t epsilon2);
extern m_newton m_r_interior_step(mpc_t z_out, mpc_t c_out, const mpc_t z_guess, const mpc_t c_guess, const mpc_t interior, int period);
extern m_newton m_r_interior(mpc_t z_out, mpc_t c_out, const mpc_t z_guess, const mpc_t c_guess, const mpc_t interior, int period, int maxsteps);

extern m_newton m_r_domain_coord_step_raw(mpc_t c_out, const mpc_t c_guess, const mpc_t domain_coord, int loperiod, int hiperiod, mpc_t z, mpc_t dc, mpc_t zp, mpc_t dcp, mpc_t f, mpc_t df, mpc_t c_new, mpc_t d, mpfr_t d2, mpfr_t epsilon2);
extern m_newton m_r_domain_coord_step(mpc_t c_out, const mpc_t c_guess, const mpc_t domain_coord, int loperiod, int hiperiod);
extern m_newton m_r_domain_coord(mpc_t c_out, const mpc_t c_guess, const mpc_t domain_coord, int loperiod, int hiperiod, int maxsteps);
extern m_newton m_r_domain_coord_dynamic_step_raw(mpc_t c_out, const mpc_t c_guess, const mpc_t domain_coord, int hiperiod, mpc_t z, mpc_t dc, mpc_t zp, mpc_t dcp, mpc_t f, mpc_t df, mpc_t c_new, mpc_t d, mpfr_t d2, mpfr_t epsilon2);
extern m_newton m_r_domain_coord_dynamic_step(mpc_t c_out, const mpc_t c_guess, const mpc_t domain_coord, int hiperiod);
extern m_newton m_r_domain_coord_dynamic(mpc_t c_out, const mpc_t c_guess, const mpc_t domain_coord, int hiperiod, int maxsteps);

extern int m_r_parent(mpq_t angle_out, mpc_t root_out, mpc_t parent_out, const mpc_t nucleus, int period, int maxsteps, int ncpus);

extern void m_r_size(mpc_t size, const mpc_t nucleus, int period);
extern int m_r_domain_size(mpfr_t size, const mpc_t nucleus, int period);
extern m_shape m_r_shape(const mpc_t nucleus, int period);
extern void m_r_shape_estimate(mpc_t shape, const mpc_t nucleus, int period);
extern m_shape m_r_shape_discriminant(const mpc_t shape);

struct m_r_exray_in;
typedef struct m_r_exray_in m_r_exray_in;
extern m_r_exray_in *m_r_exray_in_new(const mpq_t angle, int sharpness);
extern void m_r_exray_in_delete(m_r_exray_in *ray);
extern m_newton m_r_exray_in_step(m_r_exray_in *ray, int maxsteps);
extern void m_r_exray_in_get(const m_r_exray_in *ray, mpc_t c);

struct m_r_exray_out;
typedef struct m_r_exray_out m_r_exray_out;
extern m_r_exray_out *m_r_exray_out_new(const mpc_t c, int sharpness, int maxdwell, double er);
extern void m_r_exray_out_delete(m_r_exray_out *ray);
extern m_newton m_r_exray_out_step(m_r_exray_out *ray);
extern void m_r_exray_out_get(const m_r_exray_out *ray, mpc_t endpoint);
extern bool m_r_exray_out_have_bit(const m_r_exray_out *ray);
extern bool m_r_exray_out_get_bit(const m_r_exray_out *ray);
```

```
extern char *m_r_exray_out_do(const mpc_t c, int sharpness, int maxdwell, double er);

struct m_r_exray_out_perturbed;
typedef struct m_r_exray_out_perturbed m_r_exray_out_perturbed;
extern m_r_exray_out_perturbed *m_r_exray_out_perturbed_new(const mpc_t nucleus, int period, const mpc_t endpoint, int sharpness, int maxdwell, double er);
extern void m_r_exray_out_perturbed_delete(m_r_exray_out_perturbed *ray);
extern m_newton m_r_exray_out_perturbed_step(m_r_exray_out_perturbed *ray);
extern bool m_r_exray_out_perturbed_have_bit(const m_r_exray_out_perturbed *ray);
extern bool m_r_exray_out_perturbed_get_bit(const m_r_exray_out_perturbed *ray);
extern double _Complex m_r_exray_out_perturbed_get(const m_r_exray_out_perturbed *ray);
extern char *m_r_exray_out_perturbed_do(const mpc_t nucleus, int period, const mpc_t c, int sharpness, int maxdwell, double er);

struct m_r_box_period;
typedef struct m_r_box_period m_r_box_period;
extern m_r_box_period *m_r_box_period_new(const mpc_t center, const mpfr_t radius);
extern void m_r_box_period_delete(m_r_box_period *box);
extern bool m_r_box_period_step(m_r_box_period *box);
extern bool m_r_box_period_have_period(const m_r_box_period *box);
extern int m_r_box_period_get_period(const m_r_box_period *box);
extern int m_r_box_period_do(const mpc_t center, const mpfr_t radius, int maxperiod);

struct m_r_box_misiurewicz;
typedef struct m_r_box_misiurewicz m_r_box_misiurewicz;
extern m_r_box_misiurewicz *m_r_box_misiurewicz_new(const mpc_t center, const mpfr_t radius);
extern void m_r_box_misiurewicz_delete(m_r_box_misiurewicz *box);
extern bool m_r_box_misiurewicz_step(m_r_box_misiurewicz *box);
extern bool m_r_box_misiurewicz_check_periods(m_r_box_misiurewicz *box, int maxperiod);
extern int m_r_box_misiurewicz_get_preperiod(const m_r_box_misiurewicz *box);
extern int m_r_box_misiurewicz_get_period(const m_r_box_misiurewicz *box);
extern bool m_r_box_misiurewicz_do(int *preperiod, int *period, const mpc_t center, const mpfr_t radius, int maxpreperiod, int maxperiod);

struct m_r_ball_period;
typedef struct m_r_ball_period m_r_ball_period;
extern m_r_ball_period *m_r_ball_period_new(const mpc_t center, const mpfr_t radius);
extern void m_r_ball_period_delete(m_r_ball_period *ball);
extern bool m_r_ball_period_step(m_r_ball_period *ball);
extern bool m_r_ball_period_have_period(const m_r_ball_period *ball);
extern int m_r_ball_period_get_period(const m_r_ball_period *ball);
extern int m_r_ball_period_do(const mpc_t center, const mpfr_t radius, int maxperiod);

extern void m_r_external_angles(char **lo, char **hi, const mpc_t nucleus, int period, double radius, int resolution);

#ifdef __cplusplus
```

```
}
#endif

#endif
```

38 c/lib/Makefile

```
## mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
## Copyright (C) 2015-2018 Claude Heiland-Allen
## License GPL3+ http://www.gnu.org/licenses/gpl.html

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

COMPILE := $(CC) -std=c11 -Wall -Wextra -pedantic -fPIC -O3 -pipe -g -MMD -I../include -c
LINK    := $(CC) -shared -g
LIBRARY := libmandelbrot-numerics
OBJECTS := $(patsubst %.c,%.o,$(wildcard *.c))
DEPENDS := $(patsubst %.o,%.d,$(OBJECTS))

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

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

install: $(LIBRARY).a $(LIBRARY).so ../include/mandelbrot-numerics.h pkgconfig/mandelbrot-numerics.pc
	install -d "$(prefix)/include" "$(prefix)/lib" "$(prefix)/lib/pkgconfig"
	install -m 644 -t "$(prefix)/include" ../include/mandelbrot-numerics.h
	install -m 644 -t "$(prefix)/lib" $(LIBRARY).a $(LIBRARY).so
	install -m 644 -t "$(prefix)/lib/pkgconfig" pkgconfig/mandelbrot-numerics.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-numerics.pc: pkgconfig/mandelbrot-numerics.pc.in
	@echo "PC      $@" ; ( echo "prefix=$(prefix)" ; cat pkgconfig/mandelbrot-numerics.pc.in ) > pkgconfig/mandelbrot-numerics.pc || ( echo 'ERROR    ( echo "prefix=$(prefix)" ; cat pkgconfig/mandelbrot-numerics.pc.in ) > pkgconfig/mandelbrot-numerics.pc' && false )

.SUFFIXES:
.PHONY: all clean install
```

```
-include $(DEPENDS)
```

39 c/lib/m_d_attractor.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

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

extern m_newton m_d_attractor_step(double _Complex *z, double _Complex z_guess, double _Complex c, int period) {
  double _Complex zz = z_guess;
  double _Complex dzz = 1;
  for (int i = 0; i < period; ++i) {
    dzz = 2 * zz * dzz;
    zz = zz * zz + c;
  }
  if (cabs2(zz - z_guess) <= epsilon2) {
    *z = z_guess;
    return m_converged;
  }
  double _Complex z_new = z_guess - (zz - z_guess) / (dzz - 1);
  double _Complex d = z_new - z_guess;
  if (cabs2(d) <= epsilon2) {
    *z = z_new;
    return m_converged;
  }
  if (cisfinite(d)) {
    *z = z_new;
    return m_stepped;
  } else {
    *z = z_guess;
    return m_failed;
  }
}

extern m_newton m_d_attractor(double _Complex *z_out, double _Complex z_guess, double _Complex c, int period, int maxsteps) {
  m_newton result = m_failed;
  double _Complex z = z_guess;
  for (int i = 0; i < maxsteps; ++i) {
    if (m_stepped != (result = m_d_attractor_step(&z, z, c, period))) {
      break;
    }
  }
  *z_out = z;
  return result;
}
```

40 c/lib/m_d_ball_period.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html
```

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

struct m_d_ball_period {
  double _Complex c;
  double _Complex z;
  double _Complex dz;
  double r, rdz, rz, rr, Ei;
  int p;
};

extern m_d_ball_period *m_d_ball_period_new(double _Complex center, double radius) {
  m_d_ball_period *ball = (m_d_ball_period *) malloc(sizeof(*ball));
  if (! ball) {
    return 0;
  }
  ball->c = center;
  ball->r = radius;
  ball->z = 0;
  ball->rdz = 0;
  ball->rz = 0;
  ball->rr = 0;
  ball->Ei = 0;
  ball->p = 0;
  m_d_ball_period_step(ball);
  return ball;
}

extern void m_d_ball_period_delete(m_d_ball_period *ball) {
  if (ball) {
    free(ball);
  }
}

extern bool m_d_ball_period_step(m_d_ball_period *ball) {
  if (! ball) {
    return false;
  }
  ball->Ei = ball->rdz * ball->rdz + (2 * ball->rz + ball->r * (2 * ball->rdz + ball->r * ball->Ei)) * ball->Ei;
  ball->dz = 2 * ball->z * ball->dz + 1;
  ball->z = ball->z * ball->z + ball->c;
  ball->rdz = cabs(ball->dz);
  ball->rz = cabs(ball->z);
  ball->rr = ball->r * (ball->rdz + ball->r * ball->Ei);
  ball->p = ball->p + 1;
  return ball->rz - ball->rr <= 2;
}

extern bool m_d_ball_period_have_period(const m_d_ball_period *ball) {
  if (! ball) {
    return true;
  }
  return ball->rz <= ball->rr;
```

```
}

extern int m_d_ball_period_get_period(const m_d_ball_period *ball) {
  if (! ball) {
    return 0;
  }
  return ball->p;
}

extern int m_d_ball_period_do(double _Complex center, double radius, int ↙
    ↪ maxperiod) {
  m_d_ball_period *ball = m_d_ball_period_new(center, radius);
  if (! ball) {
    return 0;
  }
  int period = 0;
  for (int i = 0; i < maxperiod; ++i) {
    if (m_d_ball_period_have_period(ball)) {
      period = m_d_ball_period_get_period(ball);
      break;
    }
    if (! m_d_ball_period_step(ball)) {
      break;
    }
  }
  m_d_ball_period_delete(ball);
  return period;
}
```

41 c/lib/m_d_box_misiurewicz.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

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

static double cross(double _Complex a, double _Complex b) {
  return cimag(a) * creal(b) - creal(a) * cimag(b);
}

static bool crosses_positive_real_axis(double _Complex a, double _Complex b) {
  if (sgn(cimag(a)) != sgn(cimag(b))) {
    double _Complex d = b - a;
    int s = sgn(cimag(d));
    int t = sgn(cross(d, a));
    return s == t;
  }
  return false;
}

static bool surrounds_origin(double _Complex a, double _Complex b, double ↙
    ↪ _Complex c, double _Complex d) {
  return odd
    ( crosses_positive_real_axis(a, b)
    + crosses_positive_real_axis(b, c)
```

```
    + crosses_positive_real_axis(c, d)
    + crosses_positive_real_axis(d, a)
    );
}

struct m_d_box_misiurewicz {
  double _Complex center;
  double _Complex w, wp;
  double _Complex c[4];
  double _Complex z[4];
  int preperiod;
  int period;
};

extern m_d_box_misiurewicz *m_d_box_misiurewicz_new(double _Complex center, double radius) {
  m_d_box_misiurewicz *box = (m_d_box_misiurewicz *) malloc(sizeof(*box));
  if (! box) {
    return 0;
  }
  box->center = center;
  box->w = center;
  box->z[0] = box->c[0] = center + ((-radius) + I * (-radius));
  box->z[1] = box->c[1] = center + (( radius) + I * (-radius));
  box->z[2] = box->c[2] = center + (( radius) + I * ( radius));
  box->z[3] = box->c[3] = center + ((-radius) + I * ( radius));
  box->preperiod = 1;
  box->period = 0;
  return box;
}

extern void m_d_box_misiurewicz_delete(m_d_box_misiurewicz *box) {
  if (box) {
    free(box);
  }
}

extern bool m_d_box_misiurewicz_step(m_d_box_misiurewicz *box) {
  if (! box) {
    return false;
  }
  box->w = box->w * box->w + box->center;
  bool ok = cisfinite(box->w);
  for (int i = 0; i < 4; ++i) {
    box->z[i] = box->z[i] * box->z[i] + box->c[i];
    ok = ok && cisfinite(box->z[i]);
  }
  box->preperiod++;
  return ok;
}

extern bool m_d_box_misiurewicz_check_periods(m_d_box_misiurewicz *box, int maxperiod) {
  if (! box) {
    return true;
  }
  double _Complex z = box->w;
```

```
  for (int period = 1; period <= maxperiod; ++period)
  {
    z = z * z + box->center;
    if (surrounds_origin(box->z[0] - z, box->z[1] - z, box->z[2] - z, box->z[3] - z))
    {
      box->period = period;
      return true;
    }
  }
  return false;
}

extern int m_d_box_misiurewicz_get_period(const m_d_box_misiurewicz *box) {
  if (! box) {
    return 0;
  }
  return box->period;
}

extern int m_d_box_misiurewicz_get_preperiod(const m_d_box_misiurewicz *box) {
  if (! box) {
    return -1;
  }
  return box->preperiod;
}

extern bool m_d_box_misiurewicz_do(int *preperiod, int *period, double _Complex center, double radius, int maxpreperiod, int maxperiod) {
  m_d_box_misiurewicz *box = m_d_box_misiurewicz_new(center, radius);
  if (! box) {
    return false;
  }
  for (int i = 0; i < maxpreperiod; ++i) {
    if (m_d_box_misiurewicz_check_periods(box, maxperiod)) {
      *preperiod = m_d_box_misiurewicz_get_preperiod(box);
      *period = m_d_box_misiurewicz_get_period(box);
      m_d_box_misiurewicz_delete(box);
      return true;
    }
    if (! m_d_box_misiurewicz_step(box)) {
      break;
    }
  }
  m_d_box_misiurewicz_delete(box);
  return false;
}
```

42 c/lib/m_d_box_period.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

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

```
static double cross(double _Complex a, double _Complex b) {
  return cimag(a) * creal(b) - creal(a) * cimag(b);
}

static bool crosses_positive_real_axis(double _Complex a, double _Complex b) {
  if (sgn(cimag(a)) != sgn(cimag(b))) {
    double _Complex d = b - a;
    int s = sgn(cimag(d));
    int t = sgn(cross(d, a));
    return s == t;
  }
  return false;
}

static bool surrounds_origin(double _Complex a, double _Complex b, double _Complex c, double _Complex d) {
  return odd
    ( crosses_positive_real_axis(a, b)
    + crosses_positive_real_axis(b, c)
    + crosses_positive_real_axis(c, d)
    + crosses_positive_real_axis(d, a)
    );
}

struct m_d_box_period {
  double _Complex c[4];
  double _Complex z[4];
  int p;
};

extern m_d_box_period *m_d_box_period_new(double _Complex center, double radius) {
  m_d_box_period *box = (m_d_box_period *) malloc(sizeof(*box));
  if (! box) {
    return 0;
  }
  box->z[0] = box->c[0] = center + ((-radius) + I * (-radius));
  box->z[1] = box->c[1] = center + (( radius) + I * (-radius));
  box->z[2] = box->c[2] = center + (( radius) + I * ( radius));
  box->z[3] = box->c[3] = center + ((-radius) + I * ( radius));
  box->p = 1;
  return box;
}

extern void m_d_box_period_delete(m_d_box_period *box) {
  if (box) {
    free(box);
  }
}

extern bool m_d_box_period_step(m_d_box_period *box) {
  if (! box) {
    return false;
  }
  bool ok = true;
  for (int i = 0; i < 4; ++i) {
    box->z[i] = box->z[i] * box->z[i] + box->c[i];
```

```
    ok = ok && cisfinite(box->z[i]);
  }
  box->p = box->p + 1;
  return ok;
}

extern bool m_d_box_period_have_period(const m_d_box_period *box) {
  if (! box) {
    return true;
  }
  return surrounds_origin(box->z[0], box->z[1], box->z[2], box->z[3]);
}

extern int m_d_box_period_get_period(const m_d_box_period *box) {
  if (! box) {
    return 0;
  }
  return box->p;
}

extern int m_d_box_period_do(double _Complex center, double radius, int ↙
    ↪ maxperiod) {
  m_d_box_period *box = m_d_box_period_new(center, radius);
  if (! box) {
    return 0;
  }
  int period = 0;
  for (int i = 0; i < maxperiod; ++i) {
    if (m_d_box_period_have_period(box)) {
      period = m_d_box_period_get_period(box);
      break;
    }
    if (! m_d_box_period_step(box)) {
      break;
    }
  }
  m_d_box_period_delete(box);
  return period;
}
```

43 c/lib/m_d_domain_coord.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

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

extern m_newton m_d_domain_coord_step(double _Complex *c_out, double _Complex ↙
    ↪ c_guess, double _Complex domain_coord, int loperiod, int hiperiod) {
  double _Complex zp = 0;
  double _Complex dcp = 0;
  double _Complex z = 0;
  double _Complex dc = 0;
  for (int i = 1; i <= hiperiod; ++i) {
    dc = 2 * z * dc + 1;
```

```
      z = z * z + c_guess;
      if (i == loperiod)
      {
        zp = z;
        dcp = dc;
      }
    }
    double _Complex f = z / zp - domain_coord;
    double _Complex df = (dc * zp - z * dcp) / (zp * zp);
    if (cabs2(df) <= epsilon2) {
      *c_out = c_guess;
      return m_converged;
    }
    double _Complex c_new = c_guess - f / df;
    double _Complex d = c_new - c_guess;
    if (cabs2(d) <= epsilon2) {
      *c_out = c_new;
      return m_converged;
    }
    if (cisfinite(d)) {
      *c_out = c_new;
      return m_stepped;
    } else {
      *c_out = c_guess;
      return m_failed;
    }
  }

  extern m_newton m_d_domain_coord(double _Complex *c_out, double _Complex c_guess
      , double _Complex domain_coord, int loperiod, int hiperiod, int maxsteps)
      {
    m_newton result = m_failed;
    double _Complex c = c_guess;
    for (int i = 0; i < maxsteps; ++i) {
      if (m_stepped != (result = m_d_domain_coord_step(&c, c, domain_coord,
          loperiod, hiperiod))) {
        break;
      }
    }
    *c_out = c;
    return result;
  }
```

44 c/lib/m_d_domain_size.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

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

extern double m_d_domain_size(double _Complex nucleus, int period)
{
  double _Complex z = nucleus;
  double _Complex dc = 1;
  double zq2 = cabs2(z);
```

```
  for (int q = 2; q <= period; ++q)
  {
    dc = 2 * z * dc + 1;
    z = z * z + nucleus;
    double zp2 = cabs2(z);
    if (q < period && zp2 < zq2)
      zq2 = zp2;
  }
  return sqrt(zq2) / cabs(dc);
}

extern double m_d_filtered_domain_size(double _Complex nucleus, int period, ↙
    ↳ m_period_filter_t *filter)
{
  double _Complex z = 0;
  double _Complex dc = 0;
  double zq = 1.0/0.0;
  for (int q = 1; q <= period; ++q)
  {
    dc = 2 * z * dc + 1;
    z = z * z + nucleus;
    double zp = cabs(z);
    if (q < period && zp < zq && filter->accept(filter, q))
      zq = zp;
  }
  return zq / cabs(dc);
}
```

45 c/lib/m_d_equipotential.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2019 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

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

struct m_d_equipotential {
  int count;
  double _Complex c;
  double _Complex d;
  double _Complex z;
};

extern m_d_equipotential *m_d_equipotential_new(const double _Complex c, int ↙
    ↳ sharpness, int count) {
  m_d_equipotential *ray = malloc(sizeof(*ray));
  ray->count = count;
  ray->c = c;
  ray->d = cexp(I * twopi / sharpness);
  double _Complex z = 0;
  for (int i = 0; i < count; ++i)
  {
    z = z * z + c;
  }
  ray->z = z;
  return ray;
```

```
}

extern void m_d_equipotential_delete(m_d_equipotential *ray) {
  if (! ray) {
    return;
  }
  free(ray);
}

extern m_newton m_d_equipotential_step(m_d_equipotential *ray, int maxsteps) {
  ray->z *= ray->d;
  double _Complex target = ray->z;
  double _Complex c = ray->c;
  for (int i = 0; i < maxsteps; ++i) {
    double _Complex z = 0;
    double _Complex dc = 0;
    for (int p = 0; p < ray->count; ++p) {
      dc = 2 * z * dc + 1;
      z = z * z + c;
    }
    double _Complex c_new = c - (z - target) / dc;
    if (cisfinite(c_new)) {
      c = c_new;
    } else {
      break;
    }
  }
  if (cisfinite(c)) {
    ray->c = c;
    return m_stepped;
  } else {
    return m_failed;
  }
}

extern double _Complex m_d_equipotential_get(const m_d_equipotential *ray) {
  if (! ray) {
    return 0;
  }
  return ray->c;
}
```

46 c/lib/m_d_exray_in.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

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

struct m_d_exray_in {
  mpq_t angle;
  mpq_t one;
  int sharpness;
  double er;
  double _Complex c;
```

```
  int j;
  int k;
};

extern m_d_exray_in *m_d_exray_in_new(const mpq_t angle, int sharpness) {
  m_d_exray_in *ray = malloc(sizeof(*ray));
  mpq_init(ray->angle);
  mpq_set(ray->angle, angle);
  mpq_init(ray->one);
  mpq_set_ui(ray->one, 1, 1);
  ray->sharpness = sharpness;
  ray->er = 65536.0;
  double a = twopi * mpq_get_d(ray->angle);
  ray->c = ray->er * (cos(a) + I * sin(a));
  ray->k = 0;
  ray->j = 0;
  return ray;
}

extern void m_d_exray_in_delete(m_d_exray_in *ray) {
  if (! ray) {
    return;
  }
  mpq_clear(ray->angle);
  mpq_clear(ray->one);
  free(ray);
}

extern m_newton m_d_exray_in_step(m_d_exray_in *ray, int maxsteps) {
  if (ray->j >= ray->sharpness) {
    mpq_mul_2exp(ray->angle, ray->angle, 1);
    if (mpq_cmp_ui(ray->angle, 1, 1) >= 0) {
      mpq_sub(ray->angle, ray->angle, ray->one);
    }
    ray->k = ray->k + 1;
    ray->j = 0;
  }
  double r = pow(ray->er, pow(0.5, (ray->j + 0.5) / ray->sharpness));
  double a = twopi * mpq_get_d(ray->angle);
  double _Complex target = r * (cos(a) + I * sin(a));
  double _Complex c = ray->c;
  for (int i = 0; i < maxsteps; ++i) {
    double _Complex z = 0;
    double _Complex dc = 0;
    for (int p = 0; p <= ray->k; ++p) {
      dc = 2 * z * dc + 1;
      z = z * z + c;
    }
    double _Complex c_new = c - (z - target) / dc;
    double d2 = cabs2(c_new - c);
    if (cisfinite(c_new)) {
      c = c_new;
    } else {
      break;
    }
    if (d2 <= epsilon2) {
      break;
```

```
    }
  }
  ray->j = ray->j + 1;
  double d2 = cabs2(c - ray->c);
  if (d2 <= epsilon2) {
    ray->c = c;
    return m_converged;
  }
  if (cisfinite(c)) {
    ray->c = c;
    return m_stepped;
  } else {
    return m_failed;
  }
}

extern double _Complex m_d_exray_in_get(const m_d_exray_in *ray) {
  if (! ray) {
    return 0;
  }
  return ray->c;
}

extern double _Complex m_d_exray_in_do(const mpq_t angle, int sharpness, int
    ↪ maxsteps, int maxnewtonsteps) {
  m_d_exray_in *ray = m_d_exray_in_new(angle, sharpness);
  if (! ray) {
    return 0;
  }
  for (int i = 0; i < maxsteps; ++i) {
    if (m_stepped != m_d_exray_in_step(ray, maxnewtonsteps)) {
      break;
    }
  }
  double _Complex endpoint = m_d_exray_in_get(ray);
  m_d_exray_in_delete(ray);
  return endpoint;
}
```

47 c/lib/m_d_exray_out.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

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

static double dwell(double loger2, int n, double zmag2) {
  return n - log2(log(zmag2) / loger2);
}

struct m_d_exray_out {
  int sharpness;
  double er;
  double er2;
  double loger2;
```

```
  double _Complex c;
  double _Complex z;
  double d;
  int n;
  int bit;
};

extern m_d_exray_out *m_d_exray_out_new(double _Complex c, int sharpness, int ↙
    ↳ maxdwell) {
  double er = 65536;
  double er2 = er * er;
  int n = 0;
  double _Complex z = 0;
  for (int i = 0; i < maxdwell; ++i) {
    n = n + 1;
    z = z * z + c;
    if (cabs2(z) > er2) {
      break;
    }
  }
  if (! (cabs2(z) > er2)) {
    return 0;
  }
  m_d_exray_out *ray = malloc(sizeof(*ray));
  if (! ray) {
    return 0;
  }
  ray->sharpness = sharpness;
  ray->er = er;
  ray->er2 = er2;
  ray->loger2 = log(er2);
  ray->c = c;
  ray->z = z;
  ray->d = dwell(ray->loger2, n, cabs2(z));
  ray->n = n;
  ray->bit = -1;
  return ray;
}

extern void m_d_exray_out_delete(m_d_exray_out *ray) {
  if (ray) {
    free(ray);
  }
}

extern m_newton m_d_exray_out_step(m_d_exray_out *ray) {
  if (! ray) {
    return m_failed;
  }
  ray->bit = -1;
  ray->d -= 1.0 / ray->sharpness;
  if (ray->d <= 0) {
    return m_converged;
  }
  int m = ceil(ray->d);
  double r = pow(ray->er, pow(2, m - ray->d));
  double a = carg(ray->z) / twopi;
```

```
  double t = a - floor(a);
  if (m == ray->n) {
    double _Complex k = r * cexp(I * twopi *  t);
    double _Complex c = ray->c;
    double _Complex z = 0;
    for (int i = 0; i < 64; ++i) { // FIXME arbitrary limit
      double _Complex dc = 0;
      z = 0;
      for (int p = 0; p < m; ++p) {
        dc = 2 * z * dc + 1;
        z = z * z + c;
      }
      double _Complex c_new = c - (z - k) / dc;
      double d2 = cabs2(c_new - c);
      if (cisfinite(c_new)) {
        c = c_new;
      } else {
        break;
      }
      if (d2 <= epsilon2) {
        break;
      }
    }
    if (cisfinite(c)) {
      ray->c = c;
      ray->z = z;
      ray->d = dwell(ray->loger2, m, cabs2(z));
      return m_stepped;
    }
    return m_failed;
  } else {
    double _Complex k[2] = { r * cexp(I * pi * t), r * cexp(I * pi * (t + 1)) };
    double _Complex c[2] = { ray->c, ray->c };
    double _Complex z[2] = { 0, 0 };
    double d2[2];
    double e2[2];
    for (int i = 0; i < 64; ++i) { // FIXME arbitrary limit
      z[0] = 0;
      z[1] = 0;
      double _Complex dc[2] = { 0, 0 };
      for (int p = 0; p < m; ++p) {
        for (int w = 0; w < 2; ++w) {
          dc[w] = 2 * z[w] * dc[w] + 1;
          z[w] = z[w] * z[w] + c[w];
        }
      }
      double _Complex c_new[2];
      for (int w = 0; w < 2; ++w) {
        c_new[w] = c[w] - (z[w] - k[w]) / dc[w];
        e2[w] = cabs2(c_new[w] - c[w]);
        d2[w] = cabs2(c_new[w] - ray->c);
        c[w] = c_new[w];
      }
      if (! (e2[0] > epsilon2 && e2[1] > epsilon2)) {
        break;
      }
    }
```

```
130       if ((cisfinite(c[0]) && cisfinite(c[1]) && d2[0] <= d2[1])
           || (cisfinite(c[0]) && ! cisfinite(c[1]))) {
            ray->bit = 0;
            ray->c = c[0];
            ray->z = z[0];
135         ray->n = m;
            ray->d = dwell(ray->loger2, m, cabs2(z[0]));
            return m_stepped;
          }
          if ((cisfinite(c[0]) && cisfinite(c[1]) && d2[1] <= d2[0])
140        || (! cisfinite(c[0]) && cisfinite(c[1]))) {
            ray->bit = 1;
            ray->c = c[1];
            ray->z = z[1];
            ray->n = m;
145         ray->d = dwell(ray->loger2, m, cabs2(z[1]));
            return m_stepped;
          }
          return m_failed;
        }
150   }

      extern bool m_d_exray_out_have_bit(const m_d_exray_out * ray) {
        if (! ray) {
          return false;
155     }
        return 0 <= ray->bit;
      }

      extern bool m_d_exray_out_get_bit(const m_d_exray_out *ray) {
160     if (! ray) {
          return false;
        }
        return ray->bit;
      }
165
      extern double _Complex m_d_exray_out_get(const m_d_exray_out *ray) {
        if (! ray) {
          return 0;
        }
170     return ray->c;
      }

      extern char *m_d_exray_out_do(double _Complex c, int sharpness, int maxdwell) {
        m_d_exray_out *ray = m_d_exray_out_new(c, sharpness, maxdwell);
175     if (! ray) {
          return 0;
        }
        char *bits = malloc(maxdwell + 2);
        if (! bits) {
180       m_d_exray_out_delete(ray);
          return 0;
        }
        int n = 0;
        while (n <= maxdwell) {
185       if (m_d_exray_out_have_bit(ray)) {
            bits[n++] = '0' + m_d_exray_out_get_bit(ray);
```

```
    }
    if (m_stepped != m_d_exray_out_step(ray)) {
      break;
    }
  }
  bits[n] = 0;
  m_d_exray_out_delete(ray);
  return bits;
}
```

48 c/lib/m_d_external_angles.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2019 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

// for strdup()
#define _POSIX_C_SOURCE 200809L

#include <complex.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mandelbrot-numerics.h>
#include "m_d_util.h"

extern void m_d_external_angles(char **lo, char **hi, double _Complex nucleus, int period, double radius, int resolution)
{
  if ((! lo) || (! hi))
  {
    return;
  }
  if (period <= 1)
  {
    *lo = strdup(".(0)");
    *hi = strdup(".(1)");
    return;
  }
  double r = m_d_domain_size(nucleus, period);
  r *= radius;
  int *counts = malloc(resolution * sizeof(int));
  int maxiter = 16 * period;
  for (int t = 0; t < resolution; ++t)
  {
    double a = 2 * pi * (t + 0.5) / resolution;
    double c = cos(a);
    double s = sin(a);
    double _Complex probe = nucleus + r * (c + I * s);
    double _Complex z = 0;
    int count = maxiter;
    for (int i = 0; i < maxiter; ++i)
    {
      z = z * z + probe;
      double mz = cabs2(z);
      if (mz > 1024)
```

```
45        {
            count = i;
            break;
          }
        }
50      counts[t] = count;
    }
    char *first = 0;
    while (1)
    {
55      int mint = -1;
      int mincount = maxiter;
      for (int t = 0; t < resolution; ++t)
      {
        if (counts[t] < mincount)
60        {
            mincount = counts[t];
            mint = t;
          }
        }
65      if (mint == -1)
      {
        if (first)
          free(first);
        *lo = 0;
70      *hi = 0;
        return;
      }
      double a = 2 * pi * (mint + 0.5) / resolution;
      double c = cos(a);
75    double s = sin(a);
      double _Complex probe = nucleus + r * (c + I * s);
      char *bits = m_d_exray_out_do(probe, 16, mincount * 2);
      int bitlen = strlen(bits);
      char *angle = malloc(period + 4);
80    angle[0] = '.';
      angle[1] = '(';
      for (int i = 0; i < period; ++i)
      {
        angle[2 + i] = bits[bitlen-1 - i];
85    }
      angle[2 + period] = ')';
      angle[3 + period] = 0;
      free(bits);
      if (first)
90    {
        if (strcmp(first, angle) != 0)
        {
          if (strcmp(first, angle) < 0)
          {
95          *lo = first;
            *hi = angle;
          }
          else
          {
100         *lo = angle;
            *hi = first;
```

```
        }
        free(counts);
        return;
      }
      else
      {
        free(angle);
      }
    }
    else
    {
      first = angle;
      angle = 0;
    }
    counts[mint] = maxiter;
  }
}
```

49 c/lib/m_d_from_logistic.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

#include <mandelbrot-numerics.h>

// https://en.wikipedia.org/wiki/Talk:Logistic_map#Very_old_discussions
extern double m_d_from_logistic(double r)
{
  double t = r - 1;
  return (1 - t * t) / 4;
}
```

50 c/lib/m_d_interior.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

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

extern m_newton m_d_interior_step(double _Complex *z_out, double _Complex *c_out↙
    ↳ , double _Complex z_guess, double _Complex c_guess, double _Complex ↙
    ↳ interior, int period) {
  double _Complex c = c_guess;
  double _Complex z = z_guess;
  double _Complex dz = 1;
  double _Complex dc = 0;
  double _Complex dzdz = 0;
  double _Complex dcdz = 0;
  for (int p = 0; p < period; ++p) {
    dcdz = 2 * (z * dcdz + dc * dz);
    dzdz = 2 * (z * dzdz + dz * dz);
    dc = 2 * z * dc + 1;
    dz = 2 * z * dz;
```

```
    z = z * z + c;
  }
  double _Complex det = (dz - 1) * dcdz - dc * dzdz;
  double _Complex z_new = z_guess - (dcdz * (z - z_guess) - dc * (dz - interior)↙
      ↳ ) / det;
  double _Complex c_new = c_guess - ((dz - 1) * (dz - interior) - dzdz * (z - ↙
      ↳ z_guess)) / det;
  if (cisfinite(z_new) && cisfinite(c_new)) {
    *z_out = z_new;
    *c_out = c_new;
    if (cabs2(z_new - z_guess) <= epsilon2 && cabs2(c_new - c_guess) <= epsilon2↙
        ↳ ) {
      return m_converged;
    } else {
      return m_stepped;
    }
  } else {
    *z_out = z_guess;
    *c_out = c_guess;
    return m_failed;
  }
}

extern m_newton m_d_interior(double _Complex *z_out, double _Complex *c_out, ↙
    ↳ double _Complex z_guess, double _Complex c_guess, double _Complex interior↙
    ↳ , int period, int maxsteps) {
  m_newton result = m_failed;
  double _Complex z = z_guess;
  double _Complex c = c_guess;
  for (int i = 0; i < maxsteps; ++i) {
    if (m_stepped != (result = m_d_interior_step(&z, &c, z, c, interior, period)↙
        ↳ )) {
      break;
    }
  }
  *z_out = z;
  *c_out = c;
  return result;
}
```

51 c/lib/m_d_interior_de.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

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

extern bool m_d_interior_de(double *de_out, double _Complex *dz_out, double ↙
    ↳ _Complex z, double _Complex c, int p, int steps) {
  double _Complex z00 = 0;
  if (m_failed != m_d_attractor(&z00, z, c, p, steps)) {
    double _Complex z0 = z00;
    double _Complex dz0 = 1;
    for (int j = 0; j < p; ++j) {
      dz0 = 2 * z0 * dz0;
```

```
      z0 = z0 * z0 + c;
    }
    if (cabs2(dz0) <= 1) {
      double _Complex z1 = z00;
      double _Complex dz1 = 1;
      double _Complex dzdz1 = 0;
      double _Complex dc1 = 0;
      double _Complex dcdz1 = 0;
      for (int j = 0; j < p; ++j) {
        dcdz1 = 2 * (z1 * dcdz1 + dz1 * dc1);
        dc1 = 2 * z1 * dc1 + 1;
        dzdz1 = 2 * (dz1 * dz1 + z1 * dzdz1);
        dz1 = 2 * z1 * dz1;
        z1 = z1 * z1 + c;
      }
      *de_out = (1 - cabs2(dz1)) / cabs(dcdz1 + dzdz1 * dc1 / (1 - dz1));
      *dz_out = dz1;
      return true;
    }
  }
  return false;
}
```

52 c/lib/m_d_mat2.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

#include <mandelbrot-numerics.h>

extern void m_d_mat2_set(m_d_mat2 *o, const m_d_mat2 *m) {
  o->a = m->a;
  o->b = m->b;
  o->c = m->c;
  o->d = m->d;
}

extern void m_d_mat2_id(m_d_mat2 *o) {
  o->a = 1;
  o->b = 0;
  o->c = 0;
  o->d = 1;
}

extern double _Complex m_d_mat2_tr(const m_d_mat2 *m) {
  return m->a + m->d;
}

extern double _Complex m_d_mat2_det(const m_d_mat2 *m) {
  return m->a * m->d - m->b * m->c;
}

extern void m_d_mat2_inv(m_d_mat2 *m1, const m_d_mat2 *m) {
  double _Complex det = m_d_mat2_det(m);
  double _Complex a =  m->d / det;
  double _Complex b = -m->b / det;
```

```
  double _Complex c = -m->c / det;
  double _Complex d =  m->a / det;
  m1->a = a;
  m1->b = b;
  m1->c = c;
  m1->d = d;
}

extern void m_d_mat2_mul(m_d_mat2 *o, const m_d_mat2 *l, const m_d_mat2 *r) {
  double _Complex a, b, c, d;
  a = l->a * r->a + l->b * r->c;
  b = l->a * r->b + l->b * r->d;
  c = l->c * r->a + l->d * r->c;
  d = l->c * r->b + l->d * r->d;
  o->a = a;
  o->b = b;
  o->c = c;
  o->d = d;
}

extern void m_d_mat2_diagonalize(m_d_mat2 *p, m_d_mat2 *d, m_d_mat2 *p1, const m_d_mat2 *m) {
  double _Complex tr2 = m_d_mat2_tr(m) / 2;
  double _Complex det = m_d_mat2_det(m);
  double _Complex k = csqrt(tr2 * tr2 - det);
  double _Complex l1 = tr2 + k;
  double _Complex l2 = tr2 - k;
  d->a = l1;
  d->b = 0;
  d->c = 0;
  d->d = l2;
  if (m->b != 0) {
    p->a = m->b;
    p->b = m->b;
    p->c = l1 - m->a;
    p->d = l2 - m->a;
    m_d_mat2_inv(p1, p);
  } else if (m->c != 0) {
    p->a = l1 - m->d;
    p->b = l2 - m->d;
    p->c = m->c;
    p->d = m->c;
    m_d_mat2_inv(p1, p);
  } else {
    p1->a = p->a = 1;
    p1->b = p->b = 0;
    p1->c = p->c = 0;
    p1->d = p->d = 1;
  }
}

extern void m_d_mat2_moebius3(m_d_mat2 *m, double _Complex zero, double _Complex one, double _Complex infinity) {
  m->a = infinity * (zero - one);
  m->b = zero * (one - infinity);
  m->c = zero - one;
  m->d = one - infinity;
```

```
}

struct m_d_mat2_interp {
  m_d_mat2 fp, d, p1;
};

extern m_d_mat2_interp *m_d_mat2_interp_new(void) {
  return (m_d_mat2_interp *) malloc(sizeof(m_d_mat2_interp));
}

extern void m_d_mat2_interp_delete(m_d_mat2_interp *i) {
  free(i);
}

extern void m_d_mat2_interp_init(m_d_mat2_interp *i, const m_d_mat2 *f, const ↙
    ↳ m_d_mat2 *g) {
  m_d_mat2 f1;
  m_d_mat2_inv(&f1, f);
  m_d_mat2 f1g;
  m_d_mat2_mul(&f1g, &f1, g);
  m_d_mat2 p;
  m_d_mat2_diagonalize(&p, &i->d, &i->p1, &f1g);
  m_d_mat2_mul(&i->fp, f, &p);
}

extern void m_d_mat2_interp_do(m_d_mat2 *m, const m_d_mat2_interp *i, double t) ↙
    ↳ {
  m_d_mat2 e;
  e.a = cpow(i->d.a, t);
  e.b = 0;
  e.c = 0;
  e.d = cpow(i->d.d, t);
  m_d_mat2 fpe;
  m_d_mat2_mul(&fpe, &i->fp, &e);
  m_d_mat2_mul(m, &fpe, &i->p1);
}
```

53 c/lib/m_d_misiurewicz.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

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

extern m_newton m_d_misiurewicz_naive_step(double _Complex *c_out, double ↙
    ↳ _Complex c_guess, int preperiod, int period) {
  double _Complex z = 0;
  double _Complex dc = 0;
  double _Complex zp = 0;
  double _Complex dcp = 0;
  for (int i = 0; i < preperiod + period; ++i) {
    if (i == preperiod) {
      zp = z;
      dcp = dc;
    }
```

```
    dc = 2 * z * dc + 1;
    z = z * z + c_guess;
  }
  double _Complex f = z - zp;
  double _Complex df = dc - dcp;
  double _Complex c_new = c_guess - f / df;
  double _Complex d = c_new - c_guess;
  if (cabs2(d) <= epsilon2) {
    *c_out = c_new;
    return m_converged;
  }
  if (cisfinite(d)) {
    *c_out = c_new;
    return m_stepped;
  } else {
    *c_out = c_guess;
    return m_failed;
  }
}

extern m_newton m_d_misiurewicz_naive(double _Complex *c_out, double _Complex
    ↪ c_guess, int preperiod, int period, int maxsteps) {
  m_newton result = m_failed;
  double _Complex c = c_guess;
  for (int i = 0; i < maxsteps; ++i) {
    if (m_stepped != (result = m_d_misiurewicz_naive_step(&c, c, preperiod,
        ↪ period))) {
      break;
    }
  }
  *c_out = c;
  return result;
}

/*
(pp+p) = (pp)
(pp+p) / (pp) - 1 = 0
  quotient rule:
    f(x) = \frac{g(x)}{h(x)}
   f'(x) = \frac{g'(x)h(x) - g(x)h'(x)}{[h(x)]^2}
((pp+p)' (pp) - (pp+p) (pp)') / (pp)^2

(0 +p) /= ( 0)
(1 +p) /= ( 1)
...
(pp+p) =  (pp)

        (pp+p) - (pp)
------------------------------------ = 0
((0 +p) - ( 0))*((1 + p) - ( 1)) ...

  product rule:
    \frac{d}{dx} \left [ \prod_{i=1}^k f_i(x) \right ]
    = \sum_{i=1}^k \left(\frac{d}{dx} f_i(x) \prod_{j\ne i} f_j(x) \right)
    = \left( \prod_{i=1}^k f_i(x) \right) \left( \sum_{i=1}^k \frac{f'_i(x)}{f_i
        ↪ (x)} \right)
```

```
    h = ((0 +p) - ( 0))*((1 + p) - ( 1)) ...
    h' = h * sum (((i +p)'-(i)') / ((i +p) - (i))
    (((pp+p)' - (pp)') * h - ((pp+p) - (pp)) * h') / h^2

    */

    extern m_newton m_d_misiurewicz_step(double _Complex *c_out, double _Complex
        ↪ c_guess, int preperiod, int period) {
      // iteration
      double _Complex z = 0;
      double _Complex dc = 0;
      double _Complex zp = 0;
      double _Complex dcp = 0;
      for (int i = 0; i < period; ++i) {
        dc = 2 * z * dc + 1;
        z = z * z + c_guess;
      }
      double _Complex h = 1;
      double _Complex dh = 0;
      for (int i = 0; i < preperiod; ++i) {
        // reject lower preperiods
        double _Complex k = z - zp;
        h = h * k;
        dh = dh + (dc - dcp) / k;
        // iterate
        dc = 2 * z * dc + 1;
        z = z * z + c_guess;
        dcp = 2 * zp * dcp + 1;
        zp = zp * zp + c_guess;
      }
      // build function
      dh = dh * h;
      double _Complex g = z - zp;
      double _Complex dg = dc - dcp;
      double _Complex f = g / h;
      double _Complex df = (dg * h - g * dh) / (h * h);
      // newton step
      double _Complex c_new = c_guess - f / df;
      // check convergence
      double _Complex d = c_new - c_guess;
      if (cabs2(d) <= epsilon2) {
        *c_out = c_new;
        return m_converged;
      }
      if (cisfinite(d)) {
        *c_out = c_new;
        return m_stepped;
      } else {
        *c_out = c_guess;
        return m_failed;
      }
    }

    extern m_newton m_d_misiurewicz(double _Complex *c_out, double _Complex c_guess,
        ↪  int preperiod, int period, int maxsteps) {
      m_newton result = m_failed;
```

```
  double _Complex c = c_guess;
  for (int i = 0; i < maxsteps; ++i) {
    if (m_stepped != (result = m_d_misiurewicz_step(&c, c, preperiod, period))) ↙
        ↳ {
      break;
    }
  }
  *c_out = c;
  return result;
}
```

54 c/lib/m_d_nucleus.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

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

extern m_newton m_d_nucleus_naive_step(double _Complex *c_out, double _Complex ↙
    ↳ c_guess, int period) {
  double _Complex z = 0;
  double _Complex dc = 0;
  for (int i = 0; i < period; ++i) {
    dc = 2 * z * dc + 1;
    z = z * z + c_guess;
  }
#if 0
  if (cabs2(dc) <= epsilon2) {
    *c_out = c_guess;
    return m_converged;
  }
#endif
  double _Complex c_new = c_guess - z / dc;
  double _Complex d = c_new - c_guess;
  if (cabs2(d) <= epsilon2) {
    *c_out = c_new;
    return m_converged;
  }
  if (cisfinite(d)) {
    *c_out = c_new;
    return m_stepped;
  } else {
    *c_out = c_guess;
    return m_failed;
  }
}

extern m_newton m_d_nucleus_naive(double _Complex *c_out, double _Complex ↙
    ↳ c_guess, int period, int maxsteps) {
  m_newton result = m_failed;
  double _Complex c = c_guess;
  for (int i = 0; i < maxsteps; ++i) {
    if (m_stepped != (result = m_d_nucleus_naive_step(&c, c, period))) {
      break;
    }
```

```
  }
  *c_out = c;
  return result;
}

extern m_newton m_d_nucleus_step(double _Complex *c_out, double _Complex c_guess
    , int period) {
  double _Complex z = 0;
  double _Complex dc = 0;
  double _Complex h = 1;
  double _Complex dh = 0;
  for (int i = 1; i <= period; ++i) {
    dc = 2 * z * dc + 1;
    z = z * z + c_guess;
    // reject lower periods
    if (i < period && period % i == 0)
    {
      h = h * z;
      dh = dh + dc / z;
    }
  }
  // build function
  dh = dh * h;
  double _Complex g = z;
  double _Complex dg = dc;
  double _Complex f = g / h;
  double _Complex df = (dg * h - g * dh) / (h * h);
  // newton step
  double _Complex c_new = c_guess - f / df;
  // check convergence
  double _Complex d = c_new - c_guess;
  if (cabs2(d) <= epsilon2) {
    *c_out = c_new;
    return m_converged;
  }
  if (cisfinite(d)) {
    *c_out = c_new;
    return m_stepped;
  } else {
    *c_out = c_guess;
    return m_failed;
  }
}

extern m_newton m_d_nucleus(double _Complex *c_out, double _Complex c_guess, int
    period, int maxsteps) {
  m_newton result = m_failed;
  double _Complex c = c_guess;
  for (int i = 0; i < maxsteps; ++i) {
    if (m_stepped != (result = m_d_nucleus_step(&c, c, period))) {
      break;
    }
  }
  *c_out = c;
  return result;
}
```

55 c/lib/m_d_parent.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

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

extern int m_d_parent(mpq_t angle, double _Complex *root_out, double _Complex *parent_out, double _Complex nucleus, int period, int maxsteps) {
  switch (m_d_shape(nucleus, period)) {
    case m_cardioid: {
      // find root directly
      double _Complex z = nucleus;
      double _Complex c = nucleus;
      m_d_interior(&z, &c, z, c, 1, period, maxsteps);
      *root_out = c;
      return 0; // no parent
    }
    case m_circle: {
      // trace internal ray to near to root
      double _Complex z = nucleus;
      double _Complex c = nucleus;
      for (int step = 0; step < maxsteps - 1; ++step) {
        m_d_interior(&z, &c, z, c, (step + 0.5) / maxsteps, period, maxsteps);
      }
      double _Complex root0 = c;
      m_d_interior(&z, &c, z, c, (maxsteps - 0.5) / maxsteps, period, maxsteps);
      double _Complex root1 = c;
      // find interior coordinate of a point just past the root into the parent
      double _Complex parent_guess = 2 * root1 - root0;
      c = parent_guess;
      z = 0;
      double mz2 = 1.0 / 0.0;
      for (int p = 1; p < period; ++p) {
        z = z * z + c;
        double z2 = cabs2(z);
        if (z2 < mz2) {
          mz2 = z2;
          if (period % p == 0) {
            double _Complex w = z;
            m_d_attractor(&w, w, c, p, maxsteps);
            double _Complex dw = 1;
            for (int q = 0 ; q < p; ++q) {
              dw = 2 * w * dw;
              w = w * w + c;
            }
            if (cabs2(dw) < 1) {
              // interior to component of period p
              int den = period / p;
              int num = ((int) round(den * carg(dw) / twopi) + den) % den;
              mpq_set_si(angle, num, den);
              mpq_canonicalize(angle);
              m_d_nucleus(&c, c, p, maxsteps);
              *parent_out = c;
              double _Complex interior = cexp(I * twopi * num / (double) den);
```

```
55                m_d_interior(&w, &c, w, c, interior, p, maxsteps);
                  *root_out = c;
                  return p; // period of parent
                }
              }
            }
          }
        }
      }
      return -1; // fail
}
```

56 c/lib/m_d_shape.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

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

extern double _Complex m_d_shape_estimate(double _Complex nucleus, int period) {
  double _Complex z = nucleus;
  double _Complex dc = 1;
  double _Complex dz = 1;
  double _Complex dcdc = 0;
  double _Complex dcdz = 0;
  for (int i = 1; i < period; ++i) {
    dcdc = 2 * (z * dcdc + dc * dc);
    dcdz = 2 * (z * dcdz + dc * dz);
    dc = 2 * z * dc + 1;
    dz = 2 * z * dz;
    z = z * z + nucleus;
  }
  double _Complex shape = - (dcdc / (2 * dc) + dcdz / dz) / (dc * dz);
  return shape;
}

extern m_shape m_d_shape_discriminant(double _Complex shape) {
  if (cabs(shape) < cabs(shape - 1)) {
    return m_cardioid;
  } else {
    return m_circle;
  }
}

extern m_shape m_d_shape(double _Complex nucleus, int period) {
  return m_d_shape_discriminant(m_d_shape_estimate(nucleus, period));
}
```

57 c/lib/m_d_size.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html
```

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

extern double _Complex m_d_size(double _Complex nucleus, int period) {
  double _Complex l = 1;
  double _Complex b = 1;
  double _Complex z = 0;
  for (int i = 1; i < period; ++i) {
    z = z * z + nucleus;
    l = 2 * z * l;
    b = b + 1 / l;
  }
  return 1 / (b * l * l);
}
```

58 c/lib/m_d_to_logistic.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

#include <mandelbrot-numerics.h>

// https://en.wikipedia.org/wiki/Talk:Logistic_map#Very_old_discussions
extern double m_d_to_logistic(double c)
{
  return 1 + sqrt(1 - 4 * c);
}
```

59 c/lib/m_d_util.h

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

#ifndef M_D_UTIL_H
#define M_D_UTIL_H 1

#include <complex.h>

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

static inline bool cisfinite(double _Complex 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;

#endif
```

60 c/lib/m_r_attractor.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

#include <mandelbrot-numerics.h>

extern m_newton m_r_attractor_step_raw(mpc_t z_out, const mpc_t z_guess, const mpc_t c, int period, mpc_t z, mpc_t dz, mpc_t z_new, mpc_t d, mpfr_t d2, mpfr_t epsilon2) {
  // z = z_guess; dz = 1;
  mpc_set(z, z_guess, MPC_RNDNN);
  mpc_set_si(dz, 1, MPC_RNDNN);
  for (int i = 0; i < period; ++i) {
    // dz = 2 * z * dz;
    mpc_mul(dz, z, dz, MPC_RNDNN);
    mpc_mul_2si(dz, dz, 1, MPC_RNDNN);
    // z = z * z + c;
    mpc_sqr(z, z, MPC_RNDNN);
    mpc_add(z, z, c, MPC_RNDNN);
  }
  // z_new = z_guess - (z - z_guess) / (dz - 1);
  mpc_sub(z, z, z_guess, MPC_RNDNN);
  mpc_sub_ui(dz, dz, 1, MPC_RNDNN);
  mpc_div(z, z, dz, MPC_RNDNN);
  mpc_sub(z_new, z_guess, z, MPC_RNDNN);
  // d = z_new - z_guess;
  mpc_sub(d, z_new, z_guess, MPC_RNDNN);
  // d2 = norm(d);
  mpc_norm(d2, d, MPFR_RNDN);
  if (mpfr_lessequal_p(d2, epsilon2)) {
    mpc_set(z_out, z_new, MPC_RNDNN);
    return m_converged;
  }
  if (mpfr_number_p(mpc_realref(d)) && mpfr_number_p(mpc_imagref(d))) {
    mpc_set(z_out, z_new, MPC_RNDNN);
    return m_stepped;
  } else {
    mpc_set(z_out, z_guess, MPC_RNDNN);
    return m_failed;
  }
}

extern m_newton m_r_attractor_step(mpc_t z_out, const mpc_t z_guess, const mpc_t
```

```
    ↪ c, int period) {
  // prec
  mpfr_prec_t precr, preci, prec;
  mpc_get_prec2(&precr, &preci, z_guess);
  prec = precr > preci ? precr : preci;
  mpc_get_prec2(&precr, &preci, c);
  prec = precr > prec ? precr : prec;
  prec = preci > prec ? preci : prec;
  mpc_set_prec(z_out, prec); // FIXME might trash when z_out = z_guess
  // init
  mpc_t z, dz, z_new, d;
  mpfr_t d2, epsilon2;
  mpc_init2(z, prec);
  mpc_init2(dz, prec);
  mpc_init2(z_new, prec);
  mpc_init2(d, prec);
  mpfr_init2(d2, prec);
  mpfr_init2(epsilon2, prec);
  // epsilon
  mpfr_set_si(epsilon2, 2, MPFR_RNDN);
  mpfr_nextabove(epsilon2);
  mpfr_sub_si(epsilon2, epsilon2, 2, MPFR_RNDN);
  mpfr_sqr(epsilon2, epsilon2, MPFR_RNDN);
  // step raw
  m_newton retval = m_r_attractor_step_raw(z_out, z_guess, c, period, z, dz, ↙
    ↪ z_new, d, d2, epsilon2);
  // cleanup
  mpc_clear(z);
  mpc_clear(dz);
  mpc_clear(z_new);
  mpc_clear(d);
  mpfr_clear(d2);
  mpfr_clear(epsilon2);
  return retval;
}

extern m_newton m_r_attractor(mpc_t z_out, const mpc_t z_guess, const mpc_t c, ↙
  ↪ int period, int maxsteps) {
  m_newton result = m_failed;
  // prec
  mpfr_prec_t precr, preci, prec;
  mpc_get_prec2(&precr, &preci, z_guess);
  prec = precr > preci ? precr : preci;
  mpc_get_prec2(&precr, &preci, c);
  prec = precr > prec ? precr : prec;
  prec = preci > prec ? preci : prec;
  // init
  mpc_t z0, z, dz, z_new, d;
  mpfr_t d2, epsilon2;
  mpc_init2(z0, prec);
  mpc_init2(z, prec);
  mpc_init2(dz, prec);
  mpc_init2(z_new, prec);
  mpc_init2(d, prec);
  mpfr_init2(d2, prec);
  mpfr_init2(epsilon2, prec);
  // epsilon
```

```
  mpfr_set_si(epsilon2, 2, MPFR_RNDN);
  mpfr_nextabove(epsilon2);
  mpfr_sub_si(epsilon2, epsilon2, 2, MPFR_RNDN);
  mpfr_sqr(epsilon2, epsilon2, MPFR_RNDN);
  // z0 = z_guess;
  mpc_set(z0, z_guess, MPC_RNDNN);
  for (int i = 0; i < maxsteps; ++i) {
    if (m_stepped != (result = m_r_attractor_step_raw(z0, z0, c, period, z, dz, z_new, d, d2, epsilon2))) {
      break;
    }
  }
  // z_out = z0;
  mpc_set_prec(z_out, prec);
  mpc_set(z_out, z0, MPC_RNDNN);
  // cleanup
  mpc_clear(z0);
  mpc_clear(z);
  mpc_clear(dz);
  mpc_clear(z_new);
  mpc_clear(d);
  mpfr_clear(d2);
  mpfr_clear(epsilon2);
  return result;
}
```

61 c/lib/m_r_ball_period.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

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

struct m_r_ball_period {
  mpc_t c, z, dz;
  mpfr_t r, rdz, rz, rr, Ei, t[3];
  int p;
};

extern m_r_ball_period *m_r_ball_period_new(const mpc_t center, const mpfr_t radius) {
  m_r_ball_period *ball = (m_r_ball_period *) malloc(sizeof(*ball));
  if (! ball) {
    return 0;
  }
  int bitsr = mpfr_get_prec(mpc_realref(center));
  int bitsi = mpfr_get_prec(mpc_imagref(center));
  int bits = bitsr > bitsi ? bitsr : bitsi;
  mpc_init2(ball->c, bits); mpc_set(ball->c, center, MPC_RNDNN);
  mpc_init2(ball->z, bits); mpc_set_ui(ball->z, 0, MPC_RNDNN);
  mpc_init2(ball->dz, bits); mpc_set_ui(ball->dz, 0, MPC_RNDNN);
  mpfr_init2(ball->r, 53); mpfr_set(ball->r, radius, MPFR_RNDU);
  mpfr_init2(ball->rdz, 53); mpfr_set_ui(ball->rdz, 0, MPFR_RNDN);
  mpfr_init2(ball->rz, 53); mpfr_set_ui(ball->rz, 0, MPFR_RNDN);
  mpfr_init2(ball->rr, 53); mpfr_set_ui(ball->rr, 0, MPFR_RNDN);
```

```
  mpfr_init2(ball->Ei, 53); mpfr_set_ui(ball->Ei, 0, MPFR_RNDN);
  mpfr_init2(ball->t[0], 53); mpfr_set_ui(ball->t[0], 0, MPFR_RNDN);
  mpfr_init2(ball->t[1], 53); mpfr_set_ui(ball->t[1], 0, MPFR_RNDN);
  mpfr_init2(ball->t[2], 53); mpfr_set_ui(ball->t[2], 0, MPFR_RNDN);
  ball->p = 0;
  m_r_ball_period_step(ball);
  return ball;
}

extern void m_r_ball_period_delete(m_r_ball_period *ball) {
  if (ball) {
    mpc_clear(ball->c);
    mpc_clear(ball->z);
    mpc_clear(ball->dz);
    mpfr_clear(ball->r);
    mpfr_clear(ball->rdz);
    mpfr_clear(ball->rz);
    mpfr_clear(ball->rr);
    mpfr_clear(ball->Ei);
    mpfr_clear(ball->t[0]);
    mpfr_clear(ball->t[1]);
    mpfr_clear(ball->t[2]);
    free(ball);
  }
}

extern bool m_r_ball_period_step(m_r_ball_period *ball) {
  if (! ball) {
    return false;
  }
  // Ei = rdz * rdz + (2 * rz + r * (2 * rdz + r * Ei)) * Ei;
  mpfr_mul(ball->t[0], ball->r, ball->Ei, MPFR_RNDU);
  mpfr_mul_2ui(ball->t[1], ball->rdz, 1, MPFR_RNDU);
  mpfr_add(ball->t[2], ball->t[0], ball->t[1], MPFR_RNDU);
  mpfr_mul(ball->t[0], ball->r, ball->t[2], MPFR_RNDU);
  mpfr_mul_2ui(ball->t[1], ball->rz, 1, MPFR_RNDU);
  mpfr_add(ball->t[2], ball->t[1], ball->t[0], MPFR_RNDU);
  mpfr_mul(ball->t[0], ball->t[2], ball->Ei, MPFR_RNDU);
  mpfr_sqr(ball->t[1], ball->rdz, MPFR_RNDU);
  mpfr_add(ball->Ei, ball->t[1], ball->t[0], MPFR_RNDU);
  // dz = 2 * z * dz + 1;
  mpc_mul(ball->dz, ball->z, ball->dz, MPC_RNDNN);
  mpfr_mul_2ui(mpc_realref(ball->dz), mpc_realref(ball->dz), 1, MPFR_RNDN);
  mpfr_mul_2ui(mpc_imagref(ball->dz), mpc_imagref(ball->dz), 1, MPFR_RNDN);
  mpfr_add_ui(mpc_realref(ball->dz), mpc_realref(ball->dz), 1, MPFR_RNDN);
  // z = z * z + c;
  mpc_sqr(ball->z, ball->z, MPC_RNDNN);
  mpc_add(ball->z, ball->z, ball->c, MPC_RNDNN);
  // rdz = cabs(dz);
  mpc_abs(ball->rdz, ball->dz, MPFR_RNDU);
  // rz = cabs(z);
  mpc_abs(ball->rz, ball->z, MPFR_RNDU);
  // rr = r * (rdz + r * Ei);
  mpfr_mul(ball->t[0], ball->r, ball->Ei, MPFR_RNDU);
  mpfr_add(ball->t[1], ball->rdz, ball->t[0], MPFR_RNDU);
  mpfr_mul(ball->rr, ball->r, ball->t[1], MPFR_RNDU);
  // retval = rz - rr <= 2;
```

```
  mpfr_sub(ball->t[0], ball->rz, ball->rr, MPFR_RNDD);
  mpfr_sub_ui(ball->t[1], ball->t[0], 2, MPFR_RNDD);
  bool retval = mpfr_sgn(ball->t[1]) <= 0;
  // p = p + 1
  ball->p = ball->p + 1;
  return retval;
}

extern bool m_r_ball_period_have_period(const m_r_ball_period *ball) {
  if (! ball) {
    return true;
  }
  // rz <= rr;
  return mpfr_lessequal_p(ball->rz, ball->rr);
}

extern int m_r_ball_period_get_period(const m_r_ball_period *ball) {
  if (! ball) {
    return 0;
  }
  return ball->p;
}

extern int m_r_ball_period_do(const mpc_t center, const mpfr_t radius, int ↙
    ↪ maxperiod) {
  m_r_ball_period *ball = m_r_ball_period_new(center, radius);
  if (! ball) {
    return 0;
  }
  int period = 0;
  for (int i = 0; i < maxperiod; ++i) {
    if (m_r_ball_period_have_period(ball)) {
      period = m_r_ball_period_get_period(ball);
      break;
    }
    if (! m_r_ball_period_step(ball)) {
      break;
    }
  }
  m_r_ball_period_delete(ball);
  return period;
}
```

62 c/lib/m_r_box_misiurewicz.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

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

static void cross(mpfr_t out, const mpc_t a, const mpc_t b, mpfr_t t) {
  mpfr_mul(out, mpc_imagref(a), mpc_realref(b), MPFR_RNDN);
  mpfr_mul(t, mpc_realref(a), mpc_imagref(b), MPFR_RNDN);
  mpfr_sub(out, out, t, MPFR_RNDN);
}
```

```
static bool crosses_positive_real_axis(const mpc_t a, const mpc_t b, mpc_t d, ↙
    ↳ mpfr_t t1, mpfr_t t2) {
  if (mpfr_sgn(mpc_imagref(a)) != mpfr_sgn(mpc_imagref(b))) {
    // d = b - a;
    mpc_sub(d, b, a, MPC_RNDNN);
    // s = sgn(cimag(d));
    int s = mpfr_sgn(mpc_imagref(d));
    // t = sgn(cross(d, a));
    cross(t1, d, a, t2);
    int t = mpfr_sgn(t1);
    return s == t;
  }
  return false;
}

static bool surrounds_origin(const mpc_t a, const mpc_t b, const mpc_t c, const ↙
    ↳ mpc_t d, mpc_t e, mpfr_t t1, mpfr_t t2) {
  return odd
    ( crosses_positive_real_axis(a, b, e, t1, t2)
    + crosses_positive_real_axis(b, c, e, t1, t2)
    + crosses_positive_real_axis(c, d, e, t1, t2)
    + crosses_positive_real_axis(d, a, e, t1, t2)
    );
}

struct m_r_box_misiurewicz {
  mpc_t center;
  mpc_t w, wp;
  mpc_t c[4];
  mpc_t z[4];
  mpc_t d[4];
  int preperiod;
  int period;
  mpc_t e;
  mpfr_t t1;
  mpfr_t t2;
};

extern m_r_box_misiurewicz *m_r_box_misiurewicz_new(const mpc_t center, const ↙
    ↳ mpfr_t radius) {
  m_r_box_misiurewicz *box = (m_r_box_misiurewicz *) malloc(sizeof(*box));
  if (! box) {
    return 0;
  }
  // prec
  mpfr_prec_t precr, preci, prec;
  mpc_get_prec2(&precr, &preci, center);
  prec = precr > preci ? precr : preci;
  // init
  mpc_init2(box->center, prec);
  mpc_init2(box->w, prec);
  mpc_init2(box->wp, prec);
  for (int i = 0; i < 4; ++i) {
    mpc_init2(box->c[i], prec);
    mpc_init2(box->z[i], prec);
    mpc_init2(box->d[i], prec);
```

```
  }
  mpc_init2(box->e, prec);
  mpfr_init2(box->t1, prec);
  mpfr_init2(box->t2, prec);
  // vars
  mpc_set(box->center, center, MPC_RNDNN);
  mpc_set(box->w, center, MPC_RNDNN);
  box->preperiod = 1;
  box->period = 0;
  // box
  mpfr_sub(mpc_realref(box->c[0]), mpc_realref(center), radius, MPFR_RNDN);
  mpfr_sub(mpc_imagref(box->c[0]), mpc_imagref(center), radius, MPFR_RNDN);
  mpfr_add(mpc_realref(box->c[1]), mpc_realref(center), radius, MPFR_RNDN);
  mpfr_sub(mpc_imagref(box->c[1]), mpc_imagref(center), radius, MPFR_RNDN);
  mpfr_add(mpc_realref(box->c[2]), mpc_realref(center), radius, MPFR_RNDN);
  mpfr_add(mpc_imagref(box->c[2]), mpc_imagref(center), radius, MPFR_RNDN);
  mpfr_sub(mpc_realref(box->c[3]), mpc_realref(center), radius, MPFR_RNDN);
  mpfr_add(mpc_imagref(box->c[3]), mpc_imagref(center), radius, MPFR_RNDN);
  mpc_set(box->z[0], box->c[0], MPC_RNDNN);
  mpc_set(box->z[1], box->c[1], MPC_RNDNN);
  mpc_set(box->z[2], box->c[2], MPC_RNDNN);
  mpc_set(box->z[3], box->c[3], MPC_RNDNN);
  return box;
}

extern void m_r_box_misiurewicz_delete(m_r_box_misiurewicz *box) {
  if (box) {
    mpc_clear(box->center);
    mpc_clear(box->w);
    for (int i = 0; i < 4; ++i) {
      mpc_clear(box->c[i]);
      mpc_clear(box->z[i]);
      mpc_clear(box->d[i]);
    }
    mpc_clear(box->e);
    mpfr_clear(box->t1);
    mpfr_clear(box->t2);
    free(box);
  }
}

extern bool m_r_box_misiurewicz_step(m_r_box_misiurewicz *box) {
  if (! box) {
    return false;
  }
  bool ok = true;
  mpc_sqr(box->w, box->w, MPC_RNDNN);
  mpc_add(box->w, box->w, box->center, MPC_RNDNN);
  for (int i = 0; i < 4; ++i) {
    // box->z[i] = box->z[i] * box->z[i] + box->c[i];
    mpc_sqr(box->z[i], box->z[i], MPC_RNDNN);
    mpc_add(box->z[i], box->z[i], box->c[i], MPC_RNDNN);
    ok = ok && mpfr_number_p(mpc_realref(box->z[i])) && mpfr_number_p(↙
        ↳ mpc_imagref(box->z[i]));
  }
  box->preperiod++;
  return ok;
```

```
}

extern bool m_r_box_misiurewicz_check_periods(m_r_box_misiurewicz *box, int maxperiod) {
  // FIXME cache w orbit?  time/space trade-off...
  if (! box) {
    return true;
  }
  // wp = w
  mpc_set(box->wp, box->w, MPC_RNDNN);
  for (int period = 1; period <= maxperiod; ++period)
  {
    // wp = wp * wp + box->center;
    mpc_sqr(box->wp, box->wp, MPC_RNDNN);
    mpc_add(box->wp, box->wp, box->center, MPC_RNDNN);
    // d = z - wp
    mpc_sub(box->d[0], box->z[0], box->wp, MPC_RNDNN);
    mpc_sub(box->d[1], box->z[1], box->wp, MPC_RNDNN);
    mpc_sub(box->d[2], box->z[2], box->wp, MPC_RNDNN);
    mpc_sub(box->d[3], box->z[3], box->wp, MPC_RNDNN);
    // if d surround 0, found (pre)period
    if (surrounds_origin(box->d[0], box->d[1], box->d[2], box->d[3], box->e, box->t1, box->t2))
    {
      box->period = period;
      return true;
    }
  }
  return false;
}


extern int m_r_box_misiurewicz_get_period(const m_r_box_misiurewicz *box) {
  if (! box) {
    return 0;
  }
  return box->period;
}

extern int m_r_box_misiurewicz_get_preperiod(const m_r_box_misiurewicz *box) {
  if (! box) {
    return -1;
  }
  return box->preperiod;
}


extern bool m_r_box_misiurewicz_do(int *preperiod, int *period, const mpc_t center, const mpfr_t radius, int maxpreperiod, int maxperiod) {
  m_r_box_misiurewicz *box = m_r_box_misiurewicz_new(center, radius);
  if (! box) {
    return false;
  }
  for (int i = 0; i < maxpreperiod; ++i) {
    if (m_r_box_misiurewicz_check_periods(box, maxperiod)) {
```

```
      *preperiod = m_r_box_misiurewicz_get_preperiod(box);
      *period = m_r_box_misiurewicz_get_period(box);
      m_r_box_misiurewicz_delete(box);
      return true;
    }
    if (! m_r_box_misiurewicz_step(box)) {
      break;
    }
  }
  m_r_box_misiurewicz_delete(box);
  return false;
}
```

63 c/lib/m_r_box_period.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

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

static void cross(mpfr_t out, const mpc_t a, const mpc_t b, mpfr_t t) {
  mpfr_mul(out, mpc_imagref(a), mpc_realref(b), MPFR_RNDN);
  mpfr_mul(t, mpc_realref(a), mpc_imagref(b), MPFR_RNDN);
  mpfr_sub(out, out, t, MPFR_RNDN);
}

static bool crosses_positive_real_axis(const mpc_t a, const mpc_t b, mpc_t d, mpfr_t t1, mpfr_t t2) {
  if (mpfr_sgn(mpc_imagref(a)) != mpfr_sgn(mpc_imagref(b))) {
    // d = b - a;
    mpc_sub(d, b, a, MPC_RNDNN);
    // s = sgn(cimag(d));
    int s = mpfr_sgn(mpc_imagref(d));
    // t = sgn(cross(d, a));
    cross(t1, d, a, t2);
    int t = mpfr_sgn(t1);
    return s == t;
  }
  return false;
}

static bool surrounds_origin(const mpc_t a, const mpc_t b, const mpc_t c, const mpc_t d, mpc_t e, mpfr_t t1, mpfr_t t2) {
  return odd
    ( crosses_positive_real_axis(a, b, e, t1, t2)
    + crosses_positive_real_axis(b, c, e, t1, t2)
    + crosses_positive_real_axis(c, d, e, t1, t2)
    + crosses_positive_real_axis(d, a, e, t1, t2)
    );
}

struct m_r_box_period {
  mpc_t c[4];
  mpc_t z[4];
  int p;
```

```
  mpc_t e;
  mpfr_t t1;
  mpfr_t t2;
};

extern m_r_box_period *m_r_box_period_new(const mpc_t center, const mpfr_t ↙
    ↳ radius) {
  m_r_box_period *box = (m_r_box_period *) malloc(sizeof(*box));
  if (! box) {
    return 0;
  }
  // prec
  mpfr_prec_t precr, preci, prec;
  mpc_get_prec2(&precr, &preci, center);
  prec = precr > preci ? precr : preci;
  // init
  for (int i = 0; i < 4; ++i) {
    mpc_init2(box->c[i], prec);
    mpc_init2(box->z[i], prec);
  }
  mpc_init2(box->e, prec);
  mpfr_init2(box->t1, prec);
  mpfr_init2(box->t2, prec);
  // box
  mpfr_sub(mpc_realref(box->c[0]), mpc_realref(center), radius, MPFR_RNDN);
  mpfr_sub(mpc_imagref(box->c[0]), mpc_imagref(center), radius, MPFR_RNDN);
  mpfr_add(mpc_realref(box->c[1]), mpc_realref(center), radius, MPFR_RNDN);
  mpfr_sub(mpc_imagref(box->c[1]), mpc_imagref(center), radius, MPFR_RNDN);
  mpfr_add(mpc_realref(box->c[2]), mpc_realref(center), radius, MPFR_RNDN);
  mpfr_add(mpc_imagref(box->c[2]), mpc_imagref(center), radius, MPFR_RNDN);
  mpfr_sub(mpc_realref(box->c[3]), mpc_realref(center), radius, MPFR_RNDN);
  mpfr_add(mpc_imagref(box->c[3]), mpc_imagref(center), radius, MPFR_RNDN);
  mpc_set(box->z[0], box->c[0], MPC_RNDNN);
  mpc_set(box->z[1], box->c[1], MPC_RNDNN);
  mpc_set(box->z[2], box->c[2], MPC_RNDNN);
  mpc_set(box->z[3], box->c[3], MPC_RNDNN);
  box->p = 1;
  return box;
}

extern void m_r_box_period_delete(m_r_box_period *box) {
  if (box) {
    for (int i = 0; i < 4; ++i) {
      mpc_clear(box->c[i]);
      mpc_clear(box->z[i]);
    }
    mpc_clear(box->e);
    mpfr_clear(box->t1);
    mpfr_clear(box->t2);
    free(box);
  }
}

extern bool m_r_box_period_step(m_r_box_period *box) {
  if (! box) {
    return false;
  }
```

```
  bool ok = true;
  for (int i = 0; i < 4; ++i) {
    // box->z[i] = box->z[i] * box->z[i] + box->c[i];
    mpc_sqr(box->z[i], box->z[i], MPC_RNDNN);
    mpc_add(box->z[i], box->z[i], box->c[i], MPC_RNDNN);
    ok = ok && mpfr_number_p(mpc_realref(box->z[i])) && mpfr_number_p(
        ↪ mpc_imagref(box->z[i]));
  }
  box->p = box->p + 1;
  return ok;
}

extern bool m_r_box_period_have_period(const m_r_box_period *box) {
  if (! box) {
    return true;
  }
  m_r_box_period *ubox = (m_r_box_period *) box; // const cast
  return surrounds_origin(box->z[0], box->z[1], box->z[2], box->z[3], ubox->e,
      ↪ ubox->t1, ubox->t2);
}

extern int m_r_box_period_get_period(const m_r_box_period *box) {
  if (! box) {
    return 0;
  }
  return box->p;
}

extern int m_r_box_period_do(const mpc_t center, const mpfr_t radius, int
    ↪ maxperiod) {
  m_r_box_period *box = m_r_box_period_new(center, radius);
  if (! box) {
    return 0;
  }
  int period = 0;
  for (int i = 0; i < maxperiod; ++i) {
    if (m_r_box_period_have_period(box)) {
      period = m_r_box_period_get_period(box);
      break;
    }
    if (! m_r_box_period_step(box)) {
      break;
    }
  }
  m_r_box_period_delete(box);
  return period;
}
```

64 c/lib/m_r_domain_coord.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

#include <mandelbrot-numerics.h>

extern m_newton m_r_domain_coord_step_raw(mpc_t c_out, const mpc_t c_guess,
```

```
    ↳ const mpc_t domain_coord, int loperiod, int hiperiod, mpc_t z, mpc_t dc, ↙
    ↳ mpc_t zp, mpc_t dcp, mpc_t f, mpc_t df, mpc_t c_new, mpc_t d, mpfr_t d2, ↙
    ↳ mpfr_t epsilon2) {
  // z = 0; dc = 0; zp = 0; dcp = 0;
  mpc_set_si(z, 0, MPC_RNDNN);
  mpc_set_si(dc, 0, MPC_RNDNN);
  mpc_set_si(zp, 0, MPC_RNDNN);
  mpc_set_si(dcp, 0, MPC_RNDNN);
  for (int i = 1; i <= hiperiod; ++i) {
    // dc = 2 * z * dc + 1;
    mpc_mul(dc, z, dc, MPC_RNDNN);
    mpc_mul_2si(dc, dc, 1, MPC_RNDNN);
    mpc_add_ui(dc, dc, 1, MPC_RNDNN);
    // z = z * z + c_guess;
    mpc_sqr(z, z, MPC_RNDNN);
    mpc_add(z, z, c_guess, MPC_RNDNN);
    if (i == loperiod)
    {
      // zp = z; dcp = dc;
      mpc_set(zp, z, MPC_RNDNN);
      mpc_set(dcp, dc, MPC_RNDNN);
    }
  }
  // f = z / zp - domain_coord;
  mpc_div(f, z, zp, MPC_RNDNN);
  mpc_sub(f, f, domain_coord, MPC_RNDNN);
  // df = (dc * zp - z * dcp) / (zp * zp);
  mpc_mul(dc, dc, zp, MPC_RNDNN);
  mpc_mul(dcp, dcp, z, MPC_RNDNN);
  mpc_sqr(zp, zp, MPC_RNDNN);
  mpc_sub(z, dc, dcp, MPC_RNDNN);
  mpc_div(df, z, zp, MPC_RNDNN);
  // check |df|^2 < epsilon2
  mpc_norm(d2, df, MPFR_RNDN);
  if (mpfr_lessequal_p(d2, epsilon2)) {
    mpc_set(c_out, c_guess, MPC_RNDNN);
    return m_converged;
  }
  // c_new = c_guess - f / df;
  mpc_div(f, f, df, MPC_RNDNN);
  mpc_sub(c_new, c_guess, f, MPC_RNDNN);
  // d = c_new - c_guess;
  mpc_sub(d, c_new, c_guess, MPC_RNDNN);
  mpc_norm(d2, d, MPFR_RNDN);
  if (mpfr_lessequal_p(d2, epsilon2)) {
    mpc_set(c_out, c_new, MPC_RNDNN);
    return m_converged;
  }
  if (mpfr_number_p(mpc_realref(d)) && mpfr_number_p(mpc_imagref(d))) {
    mpc_set(c_out, c_new, MPC_RNDNN);
    return m_stepped;
  } else {
    mpc_set(c_out, c_guess, MPC_RNDNN);
    return m_failed;
  }
}
```

```
extern m_newton m_r_domain_coord_step(mpc_t c_out, const mpc_t c_guess, const mpc_t domain_coord, int loperiod, int hiperiod) {
  // prec
  mpfr_prec_t precr, preci, prec;
  mpc_get_prec2(&precr, &preci, c_guess);
  prec = precr > preci ? precr : preci;
  mpc_set_prec(c_out, prec); // FIXME might trash when c_out = c_guess
  // init
  mpc_t z, dc, c_new, d, zp, dcp, f, df;
  mpfr_t d2, epsilon2;
  mpc_init2(z, prec);
  mpc_init2(dc, prec);
  mpc_init2(c_new, prec);
  mpc_init2(d, prec);
  mpc_init2(zp, prec);
  mpc_init2(dcp, prec);
  mpc_init2(f, prec);
  mpc_init2(df, prec);
  mpfr_init2(d2, prec);
  mpfr_init2(epsilon2, prec);
  // epsilon
  mpfr_set_si(epsilon2, 2, MPFR_RNDN);
  mpfr_nextabove(epsilon2);
  mpfr_sub_si(epsilon2, epsilon2, 2, MPFR_RNDN);
  mpfr_sqr(epsilon2, epsilon2, MPFR_RNDN);
  // step raw
  m_newton retval = m_r_domain_coord_step_raw(c_out, c_guess, domain_coord, loperiod, hiperiod, z, dc, zp, dcp, f, df, c_new, d, d2, epsilon2);
  // cleanup
  mpc_clear(z);
  mpc_clear(dc);
  mpc_clear(c_new);
  mpc_clear(d);
  mpc_clear(zp);
  mpc_clear(dcp);
  mpc_clear(f);
  mpc_clear(df);
  mpfr_clear(d2);
  mpfr_clear(epsilon2);
  return retval;
}

extern m_newton m_r_domain_coord(mpc_t c_out, const mpc_t c_guess, const mpc_t domain_coord, int loperiod, int hiperiod, int maxsteps) {
  m_newton result = m_failed;
  // prec
  mpfr_prec_t precr, preci, prec;
  mpc_get_prec2(&precr, &preci, c_guess);
  prec = precr > preci ? precr : preci;
  // init
  mpc_t c, z, dc, c_new, d, zp, dcp, f, df;
  mpfr_t d2, epsilon2;
  mpc_init2(c, prec);
  mpc_init2(z, prec);
  mpc_init2(dc, prec);
  mpc_init2(c_new, prec);
  mpc_init2(d, prec);
```

```
  mpc_init2(zp, prec);
  mpc_init2(dcp, prec);
  mpc_init2(f, prec);
  mpc_init2(df, prec);
  mpfr_init2(d2, prec);
  mpfr_init2(epsilon2, prec);
  // epsilon
  mpfr_set_si(epsilon2, 2, MPFR_RNDN);
  mpfr_nextabove(epsilon2);
  mpfr_sub_si(epsilon2, epsilon2, 2, MPFR_RNDN);
  mpfr_sqr(epsilon2, epsilon2, MPFR_RNDN);
  // c = c_guess
  mpc_set(c, c_guess, MPC_RNDNN);
  for (int i = 0; i < maxsteps; ++i) {
    if (m_stepped != (result = m_r_domain_coord_step_raw(c, c, domain_coord, ↙
        ↳ loperiod, hiperiod, z, dc, zp, dcp, f, df, c_new, d, d2, epsilon2))) {
      break;
    }
  }
  // c_out = c;
  mpc_set_prec(c_out, prec);
  mpc_set(c_out, c, MPC_RNDNN);
  // cleanup
  mpc_clear(c);
  mpc_clear(z);
  mpc_clear(dc);
  mpc_clear(c_new);
  mpc_clear(d);
  mpc_clear(zp);
  mpc_clear(dcp);
  mpc_clear(f);
  mpc_clear(df);
  mpfr_clear(d2);
  mpfr_clear(epsilon2);
  return result;
}

extern m_newton m_r_domain_coord_dynamic_step_raw(mpc_t c_out, const mpc_t ↙
    ↳ c_guess, const mpc_t domain_coord, int hiperiod, mpc_t z, mpc_t dc, mpc_t ↙
    ↳ zp, mpc_t dcp, mpc_t f, mpc_t df, mpc_t c_new, mpc_t d, mpfr_t d2, mpfr_t ↙
    ↳ epsilon2) {
  // z = 0; dc = 0; zp = 0; dcp = 0;
  mpc_set_si(z, 0, MPC_RNDNN);
  mpc_set_si(dc, 0, MPC_RNDNN);
  mpc_set_si(zp, 0, MPC_RNDNN);
  mpc_set_si(dcp, 0, MPC_RNDNN);
  mpfr_set_d(mpc_imagref(f), 1.0/0.0, MPFR_RNDN);
  for (int i = 1; i <= hiperiod; ++i) {
    // dc = 2 * z * dc + 1;
    mpc_mul(dc, z, dc, MPC_RNDNN);
    mpc_mul_2si(dc, dc, 1, MPC_RNDNN);
    mpc_add_ui(dc, dc, 1, MPC_RNDNN);
    // z = z * z + c_guess;
    mpc_sqr(z, z, MPC_RNDNN);
    mpc_add(z, z, c_guess, MPC_RNDNN);
    mpc_norm(mpc_realref(f), z, MPFR_RNDN);
    if (mpfr_less_p(mpc_realref(f), mpc_imagref(f)) && i < hiperiod)
```

```
    {
      // zp = z; dcp = dc;
      mpfr_set(mpc_imagref(f), mpc_realref(f), MPFR_RNDN);
      mpc_set(zp, z, MPC_RNDNN);
      mpc_set(dcp, dc, MPC_RNDNN);
    }
  }
  // f = z / zp - domain_coord;
  mpc_div(f, z, zp, MPC_RNDNN);
  mpc_sub(f, f, domain_coord, MPC_RNDNN);
  // df = (dc * zp - z * dcp) / (zp * zp);
  mpc_mul(dc, dc, zp, MPC_RNDNN);
  mpc_mul(dcp, dcp, z, MPC_RNDNN);
  mpc_sqr(zp, zp, MPC_RNDNN);
  mpc_sub(z, dc, dcp, MPC_RNDNN);
  mpc_div(df, z, zp, MPC_RNDNN);
  // check |df|^2 < epsilon2
  mpc_norm(d2, df, MPFR_RNDN);
  if (mpfr_lessequal_p(d2, epsilon2)) {
    mpc_set(c_out, c_guess, MPC_RNDNN);
    return m_converged;
  }
  // c_new = c_guess - f / df;
  mpc_div(f, f, df, MPC_RNDNN);
  mpc_sub(c_new, c_guess, f, MPC_RNDNN);
  // d = c_new - c_guess;
  mpc_sub(d, c_new, c_guess, MPC_RNDNN);
  mpc_norm(d2, d, MPFR_RNDN);
  if (mpfr_lessequal_p(d2, epsilon2)) {
    mpc_set(c_out, c_new, MPC_RNDNN);
    return m_converged;
  }
  if (mpfr_number_p(mpc_realref(d)) && mpfr_number_p(mpc_imagref(d))) {
    mpc_set(c_out, c_new, MPC_RNDNN);
    return m_stepped;
  } else {
    mpc_set(c_out, c_guess, MPC_RNDNN);
    return m_failed;
  }
}

extern m_newton m_r_domain_coord_dynamic_step(mpc_t c_out, const mpc_t c_guess, ↙
    ↳ const mpc_t domain_coord, int hiperiod) {
  // prec
  mpfr_prec_t precr, preci, prec;
  mpc_get_prec2(&precr, &preci, c_guess);
  prec = precr > preci ? precr : preci;
  mpc_set_prec(c_out, prec); // FIXME might trash when c_out = c_guess
  // init
  mpc_t z, dc, c_new, d, zp, dcp, f, df;
  mpfr_t d2, epsilon2;
  mpc_init2(z, prec);
  mpc_init2(dc, prec);
  mpc_init2(c_new, prec);
  mpc_init2(d, prec);
  mpc_init2(zp, prec);
  mpc_init2(dcp, prec);
```

```
225   mpc_init2(f, prec);
      mpc_init2(df, prec);
      mpfr_init2(d2, prec);
      mpfr_init2(epsilon2, prec);
      // epsilon
230   mpfr_set_si(epsilon2, 2, MPFR_RNDN);
      mpfr_nextabove(epsilon2);
      mpfr_sub_si(epsilon2, epsilon2, 2, MPFR_RNDN);
      mpfr_sqr(epsilon2, epsilon2, MPFR_RNDN);
      // step raw
235   m_newton retval = m_r_domain_coord_dynamic_step_raw(c_out, c_guess, ↙
          ↳ domain_coord, hiperiod, z, dc, zp, dcp, f, df, c_new, d, d2, epsilon2);
      // cleanup
      mpc_clear(z);
      mpc_clear(dc);
      mpc_clear(c_new);
240   mpc_clear(d);
      mpc_clear(zp);
      mpc_clear(dcp);
      mpc_clear(f);
      mpc_clear(df);
245   mpfr_clear(d2);
      mpfr_clear(epsilon2);
      return retval;
    }

250 extern m_newton m_r_domain_coord_dynamic(mpc_t c_out, const mpc_t c_guess, const↙
        ↳ mpc_t domain_coord, int hiperiod, int maxsteps) {
      m_newton result = m_failed;
      // prec
      mpfr_prec_t precr, preci, prec;
      mpc_get_prec2(&precr, &preci, c_guess);
255   prec = precr > preci ? precr : preci;
      // init
      mpc_t c, z, dc, c_new, d, zp, dcp, f, df;
      mpfr_t d2, epsilon2;
      mpc_init2(c, prec);
260   mpc_init2(z, prec);
      mpc_init2(dc, prec);
      mpc_init2(c_new, prec);
      mpc_init2(d, prec);
      mpc_init2(zp, prec);
265   mpc_init2(dcp, prec);
      mpc_init2(f, prec);
      mpc_init2(df, prec);
      mpfr_init2(d2, prec);
      mpfr_init2(epsilon2, prec);
270   // epsilon
      mpfr_set_si(epsilon2, 2, MPFR_RNDN);
      mpfr_nextabove(epsilon2);
      mpfr_sub_si(epsilon2, epsilon2, 2, MPFR_RNDN);
      mpfr_sqr(epsilon2, epsilon2, MPFR_RNDN);
275   // c = c_guess
      mpc_set(c, c_guess, MPC_RNDNN);
      for (int i = 0; i < maxsteps; ++i) {
        if (m_stepped != (result = m_r_domain_coord_dynamic_step_raw(c, c, ↙
            ↳ domain_coord, hiperiod, z, dc, zp, dcp, f, df, c_new, d, d2, epsilon2)↙
```

```
        ↪ )) {
      break;
    }
  }
  // c_out = c;
  mpc_set_prec(c_out, prec);
  mpc_set(c_out, c, MPC_RNDNN);
  // cleanup
  mpc_clear(c);
  mpc_clear(z);
  mpc_clear(dc);
  mpc_clear(c_new);
  mpc_clear(d);
  mpc_clear(zp);
  mpc_clear(dcp);
  mpc_clear(f);
  mpc_clear(df);
  mpfr_clear(d2);
  mpfr_clear(epsilon2);
  return result;
}
```

65 c/lib/m_r_domain_size.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

#include <mandelbrot-numerics.h>

extern int m_r_domain_size(mpfr_t size, const mpc_t nucleus, int period) {
  // prec
  int partial = 1;
  mpfr_prec_t precr, preci, prec;
  mpc_get_prec2(&precr, &preci, nucleus);
  prec = precr > preci ? precr : preci;
  // init
#define CVARS z, dc
#define RVARS zq2, zp2
  mpc_t CVARS;
  mpfr_t RVARS;
  mpfr_inits2(prec, RVARS, (mpfr_ptr) 0);
  mpc_init2(z, prec);
  mpc_init2(dc, prec);
  // z = nucleus; dc = 1; zq2 = norm(z);
  mpc_set(z, nucleus, MPC_RNDNN);
  mpc_set_si(dc, 1, MPC_RNDNN);
  mpc_norm(zq2, z, MPFR_RNDN);
  for (int q = 2; q <= period; ++q) {
    // dc = 2 * z * dc + 1;
    mpc_mul(dc, z, dc, MPC_RNDNN);
    mpc_mul_2si(dc, dc, 1, MPC_RNDNN);
    mpc_add_ui(dc, dc, 1, MPC_RNDNN);
    // z = z * z + nucleus;
    mpc_sqr(z, z, MPC_RNDNN);
    mpc_add(z, z, nucleus, MPC_RNDNN);
    // zp2 = norm(z);
```

```
      mpc_norm(zp2, z, MPFR_RNDN);
      if (q < period && mpfr_less_p(zp2, zq2)) {
        partial = q;
        mpfr_set(zq2, zp2, MPFR_RNDN);
      }
    }
    // size = sqrt(zq2 / norm(dc));
    mpfr_set_prec(size, prec);
    mpc_norm(zp2, dc, MPFR_RNDN);
    mpfr_div(size, zq2, zp2, MPFR_RNDN);
    mpfr_sqrt(size, size, MPFR_RNDN);
    // cleanup
    mpc_clear(z);
    mpc_clear(dc);
    mpfr_clears(RVARS, (mpfr_ptr) 0);
#undef CVARS
#undef RVARS
    return partial;
}
```

66 c/lib/m_r_exray_in.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

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

struct m_r_exray_in {
  mpq_t angle;
  mpq_t one;
  int sharpness;
  double er;
  mpfr_prec_t prec;
  mpc_t c0, c, c_new, target, z, dc, d;
  mpfr_t d2, epsilon2, epsilon256;
  int j;
  int k;
};

extern m_r_exray_in *m_r_exray_in_new(const mpq_t angle, int sharpness) {
  m_r_exray_in *ray = malloc(sizeof(*ray));
  mpq_init(ray->angle);
  mpq_set(ray->angle, angle);
  mpq_init(ray->one);
  mpq_set_ui(ray->one, 1, 1);
  ray->sharpness = sharpness;
  ray->er = 65536.0;
  ray->prec = 53;
  mpc_init2(ray->c0, ray->prec);
  mpc_init2(ray->c, ray->prec);
  mpc_init2(ray->c_new, ray->prec);
  mpc_init2(ray->target, ray->prec);
  mpc_init2(ray->z, ray->prec);
  mpc_init2(ray->dc, ray->prec);
  mpc_init2(ray->d, ray->prec);
```

```
  mpfr_init2(ray->d2, ray->prec);
  mpfr_init2(ray->epsilon2, ray->prec);
  mpfr_init2(ray->epsilon256, ray->prec);
  double a = twopi * mpq_get_d(ray->angle);
  // c0 = er * (cos(a) + I * sin(a));
  mpc_set_d_d(ray->c0, ray->er * cos(a), ray->er * sin(a), MPC_RNDNN);
  ray->k = 0;
  ray->j = 0;
  // epsilon
  mpfr_set_si(ray->epsilon2, 2, MPFR_RNDN);
  mpfr_nextabove(ray->epsilon2);
  mpfr_sub_si(ray->epsilon2, ray->epsilon2, 2, MPFR_RNDN);
  mpfr_sqr(ray->epsilon2, ray->epsilon2, MPFR_RNDN);
  mpfr_set_si(ray->epsilon256, 256, MPFR_RNDN);
  mpfr_nextabove(ray->epsilon256);
  mpfr_sub_si(ray->epsilon256, ray->epsilon256, 256, MPFR_RNDN);
  mpfr_sqr(ray->epsilon256, ray->epsilon256, MPFR_RNDN);
  return ray;
}

extern void m_r_exray_in_delete(m_r_exray_in *ray) {
  if (! ray) {
    return;
  }
  mpq_clear(ray->angle);
  mpq_clear(ray->one);
  mpc_clear(ray->c0);
  mpc_clear(ray->c);
  mpc_clear(ray->c_new);
  mpc_clear(ray->target);
  mpc_clear(ray->z);
  mpc_clear(ray->dc);
  mpc_clear(ray->d);
  mpfr_clear(ray->d2);
  mpfr_clear(ray->epsilon2);
  mpfr_clear(ray->epsilon256);
  free(ray);
}

static void m_r_exray_in_bump_prec(m_r_exray_in *ray) {
  ray->prec += 16;
  mpfr_prec_round(mpc_realref(ray->c0), ray->prec, MPFR_RNDN);
  mpfr_prec_round(mpc_imagref(ray->c0), ray->prec, MPFR_RNDN);
  mpc_set_prec(ray->c, ray->prec);
  mpc_set_prec(ray->c_new, ray->prec);
  mpc_set_prec(ray->target, ray->prec);
  mpc_set_prec(ray->z, ray->prec);
  mpc_set_prec(ray->dc, ray->prec);
  mpc_set_prec(ray->d, ray->prec);
  mpfr_set_prec(ray->d2, ray->prec);
  mpfr_set_prec(ray->epsilon2, ray->prec);
  mpfr_set_prec(ray->epsilon256, ray->prec);
  // epsilon
  mpfr_set_si(ray->epsilon2, 2, MPFR_RNDN);
  mpfr_nextabove(ray->epsilon2);
  mpfr_sub_si(ray->epsilon2, ray->epsilon2, 2, MPFR_RNDN);
  mpfr_sqr(ray->epsilon2, ray->epsilon2, MPFR_RNDN);
```

```
  mpfr_set_si(ray->epsilon256, 256, MPFR_RNDN);
  mpfr_nextabove(ray->epsilon256);
  mpfr_sub_si(ray->epsilon256, ray->epsilon256, 256, MPFR_RNDN);
  mpfr_sqr(ray->epsilon256, ray->epsilon256, MPFR_RNDN);
}

extern m_newton m_r_exray_in_step(m_r_exray_in *ray, int maxsteps) {
  if (ray->j >= ray->sharpness) {
    mpq_mul_2exp(ray->angle, ray->angle, 1);
    if (mpq_cmp_ui(ray->angle, 1, 1) >= 0) {
      mpq_sub(ray->angle, ray->angle, ray->one);
    }
    ray->k = ray->k + 1;
    ray->j = 0;
  }
  int bumps = 0;
restart:
  if (! (bumps < 64)) {
    return m_failed;
  }
  double r = pow(ray->er, pow(0.5, (ray->j + 0.5) / ray->sharpness));
  double a = twopi * mpq_get_d(ray->angle);
  mpc_set_d_d(ray->target, r * cos(a), r * sin(a), MPC_RNDNN);
  // c = c0;
  mpc_set(ray->c, ray->c0, MPC_RNDNN);
  for (int i = 0; i < maxsteps; ++i) {
    // z = 0; dc = 0;
    mpc_set_si(ray->z, 0, MPC_RNDNN);
    mpc_set_si(ray->dc, 0, MPC_RNDNN);
    for (int p = 0; p <= ray->k; ++p) {
      // dc = 2 * z * dc + 1;
      mpc_mul(ray->dc, ray->z, ray->dc, MPC_RNDNN);
      mpc_mul_2si(ray->dc, ray->dc, 1, MPC_RNDNN);
      mpc_add_ui(ray->dc, ray->dc, 1, MPC_RNDNN);
      // z = z * z + c;
      mpc_sqr(ray->z, ray->z, MPC_RNDNN);
      mpc_add(ray->z, ray->z, ray->c, MPC_RNDNN);
    }
    // c_new = c - (z - target) / dc;
    mpc_sub(ray->z, ray->z, ray->target, MPC_RNDNN);
    mpc_div(ray->z, ray->z, ray->dc, MPC_RNDNN);
    mpc_sub(ray->c_new, ray->c, ray->z, MPC_RNDNN);
    if (mpfr_regular_p(mpc_realref(ray->c_new)) && mpfr_regular_p(mpc_imagref(↙
        ↳ ray->c_new))) {
      // d2 = norm(c_new - c);
      mpc_sub(ray->d, ray->c_new, ray->c, MPC_RNDNN);
      mpc_norm(ray->d2, ray->d, MPFR_RNDN);
      // c = c_new;
      mpc_set(ray->c, ray->c_new, MPC_RNDNN);
      if (mpfr_lessequal_p(ray->d2, ray->epsilon2)) {
        break;
      }
    } else {
      break;
    }
  }
  ray->j = ray->j + 1;
```

```
  if (mpfr_regular_p(mpc_realref(ray->c)) && mpfr_regular_p(mpc_imagref(ray->c))↙
      ↳ ) {
    // d2 = norm(c - c0);
    mpc_sub(ray->d, ray->c, ray->c0, MPC_RNDNN);
    mpc_norm(ray->d2, ray->d, MPFR_RNDN);
    if (mpfr_lessequal_p(ray->d2, ray->epsilon256)) {
      m_r_exray_in_bump_prec(ray);
      bumps++;
      goto restart;
    } else {
      // c0 = c;
      mpc_set(ray->c0, ray->c, MPC_RNDNN);
      return m_stepped;
    }
  } else {
    return m_failed;
  }
}

extern void m_r_exray_in_get(const m_r_exray_in *ray, mpc_t c) {
  if (! ray) {
    return;
  }
  mpc_set_prec(c, ray->prec);
  mpc_set(c, ray->c0, MPC_RNDNN);
}

extern void m_r_exray_in_get_r(const m_r_exray_in *ray, mpfr_t x, mpfr_t y) {
  if (! ray) {
    return;
  }
  mpfr_set_prec(x, ray->prec);
  mpfr_set_prec(y, ray->prec);
  mpfr_set(x, mpc_realref(ray->c0), MPFR_RNDN);
  mpfr_set(y, mpc_imagref(ray->c0), MPFR_RNDN);
}
```

67 c/lib/m_r_exray_out.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

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

static inline bool mpc_finite_p(const mpc_t z) {
  return mpfr_number_p(mpc_realref(z)) && mpfr_number_p(mpc_imagref(z));
}

static double dwell(double loger2, int n, double zmag2) {
  return n - log2(log(zmag2) / loger2);
}

struct m_r_exray_out {
  int sharpness;
  mpfr_t er;
```

```
  mpfr_t er2;
  double loger2;
  mpc_t c;
  mpc_t z;
  double d;
  int n;
  int bit;
  mpfr_t z2;
  double erd;
  mpc_t k[2];
  mpc_t cn[2];
  mpc_t zn[2];
  mpc_t dcn[2];
  mpc_t c_new[2];
  mpfr_t d2[2];
};

extern m_r_exray_out *m_r_exray_out_new(const mpc_t c, int sharpness, int
    ↪ maxdwell, double er) {
  m_r_exray_out *ray = malloc(sizeof(*ray));
  if (! ray) {
    return 0;
  }
  mpfr_prec_t precr = mpfr_get_prec(mpc_realref(c));
  mpfr_prec_t preci = mpfr_get_prec(mpc_imagref(c));
  mpfr_prec_t prec = precr > preci ? precr : preci;
  mpfr_init2(ray->er, 53);
  mpfr_init2(ray->er2, 53);
  mpc_init2(ray->c, prec);
  mpc_init2(ray->z, prec);
  mpfr_init2(ray->z2, 53);
  mpc_init2(ray->k    [0], 53);
  mpc_init2(ray->cn   [0], prec);
  mpc_init2(ray->zn   [0], prec);
  mpc_init2(ray->dcn  [0], prec);
  mpc_init2(ray->c_new[0], prec);
  mpfr_init2(ray->d2  [0], 53);
  mpc_init2(ray->k    [1], 53);
  mpc_init2(ray->cn   [1], prec);
  mpc_init2(ray->zn   [1], prec);
  mpc_init2(ray->dcn  [1], prec);
  mpc_init2(ray->c_new[1], prec);
  mpfr_init2(ray->d2  [1], 53);

  double er2 = er * er;
  mpfr_set_d(ray->er, er, MPFR_RNDN);
  mpfr_set_d(ray->er2, er2, MPFR_RNDN);
  ray->erd = er;
  ray->loger2 = log(er2);
  int n = 0;
  mpc_set_ui_ui(ray->z, 0, 0, MPC_RNDNN);
  for (int i = 0; i < maxdwell; ++i) {
    n = n + 1;
    mpc_sqr(ray->z, ray->z, MPC_RNDNN);
    mpc_add(ray->z, ray->z, c, MPC_RNDNN);
    mpc_norm(ray->z2, ray->z, MPFR_RNDN);
    if (mpfr_greater_p(ray->z2, ray->er2)) {
```

```
75          break;
          }
        }
        if (! (mpfr_greater_p(ray->z2, ray->er2))) {
          // FIXME cleanup
80        return 0;
        }
        ray->sharpness = sharpness;
        mpc_set(ray->c, c, MPC_RNDNN);
        ray->d = dwell(ray->loger2, n, mpfr_get_d(ray->z2, MPFR_RNDN));
85      ray->n = n;
        ray->bit = -1;
        return ray;
      }

90    extern void m_r_exray_out_delete(m_r_exray_out *ray) {
        if (ray) {
          mpfr_clear(ray->er);
          mpfr_clear(ray->er2);
          mpc_clear(ray->c);
95        mpc_clear(ray->z);
          mpfr_clear(ray->z2);
          mpc_clear(ray->k    [0]);
          mpc_clear(ray->cn   [0]);
          mpc_clear(ray->zn   [0]);
100       mpc_clear(ray->dcn  [0]);
          mpc_clear(ray->c_new[0]);
          mpfr_clear(ray->d2  [0]);
          mpc_clear(ray->k    [1]);
          mpc_clear(ray->cn   [1]);
105       mpc_clear(ray->zn   [1]);
          mpc_clear(ray->dcn  [1]);
          mpc_clear(ray->c_new[1]);
          mpfr_clear(ray->d2  [1]);
          free(ray);
110     }
      }

      extern m_newton m_r_exray_out_step(m_r_exray_out *ray) {
        if (! ray) {
115       return m_failed;
        }
        ray->bit = -1;
        ray->d -= 1.0 / ray->sharpness;
        if (ray->d <= 0) {
120       return m_converged;
        }
        int m = ceil(ray->d);
        double r = pow(ray->erd, pow(2, m - ray->d));
        mpc_arg(ray->z2, ray->z, MPFR_RNDN);
125     double a = mpfr_get_d(ray->z2, MPFR_RNDN) / twopi;
        double t = a - floor(a);

        if (m == ray->n) {
          mpc_set_dc(ray->k[0], r * cexp(I * twopi *  t), MPC_RNDNN);
130       mpc_set(ray->cn[0], ray->c, MPC_RNDNN);
          for (int i = 0; i < 8; ++i) { // FIXME arbitrary limit
```

```
        mpc_set_ui_ui(ray->zn[0], 0, 0, MPC_RNDNN);
        mpc_set_ui_ui(ray->dcn[0], 0, 0, MPC_RNDNN);
        for (int p = 0; p < m; ++p) {
          // dc = 2 * z * dc + 1;
          mpc_mul(ray->dcn[0], ray->zn[0], ray->dcn[0], MPC_RNDNN);
          mpfr_mul_2exp(mpc_realref(ray->dcn[0]), mpc_realref(ray->dcn[0]), 1, ↙
              ↳ MPFR_RNDN);
          mpfr_mul_2exp(mpc_imagref(ray->dcn[0]), mpc_imagref(ray->dcn[0]), 1, ↙
              ↳ MPFR_RNDN);
          mpfr_add_ui(mpc_realref(ray->dcn[0]), mpc_realref(ray->dcn[0]), 1, ↙
              ↳ MPFR_RNDN);
          // z = z * z + c;
          mpc_sqr(ray->zn[0], ray->zn[0], MPC_RNDNN);
          mpc_add(ray->zn[0], ray->zn[0], ray->cn[0], MPC_RNDNN);
        }
        // c_new = c - (z - k) / dc;
        mpc_sub(ray->c_new[0], ray->zn[0], ray->k[0], MPC_RNDNN);
        mpc_div(ray->c_new[0], ray->c_new[0], ray->dcn[0], MPC_RNDNN);
        mpc_sub(ray->c_new[0], ray->cn[0], ray->c_new[0], MPC_RNDNN);
        // c = c_new;
        mpc_set(ray->cn[0], ray->c_new[0], MPC_RNDNN);
/*
        double d2 = cabs2(c_new - c);
        if (mpc_finite_p(c_new)) {
          c = c_new;
        } else {
          break;
        }
        if (d2 <= epsilon2) {
          break;
        }
*/
      }
      if (mpc_finite_p(ray->cn[0])) {
        mpc_set(ray->c, ray->cn[0], MPC_RNDNN);
        mpc_set(ray->z, ray->zn[0], MPC_RNDNN);
        mpc_norm(ray->z2, ray->z, MPFR_RNDN);
        ray->d = dwell(ray->loger2, m, mpfr_get_d(ray->z2, MPFR_RNDN));
        return m_stepped;
      }
      return m_failed;

    } else {
      mpc_set_dc(ray->k[0], r * cexp(I * pi *  t     ), MPC_RNDNN);
      mpc_set_dc(ray->k[1], r * cexp(I * pi * (t + 1)), MPC_RNDNN);
      mpc_set(ray->cn[0], ray->c, MPC_RNDNN);
      mpc_set(ray->cn[1], ray->c, MPC_RNDNN);
      for (int i = 0; i < 8; ++i) { // FIXME arbitrary limit
        mpc_set_ui_ui(ray->zn[0], 0, 0, MPC_RNDNN);
        mpc_set_ui_ui(ray->zn[1], 0, 0, MPC_RNDNN);
        mpc_set_ui_ui(ray->dcn[0], 0, 0, MPC_RNDNN);
        mpc_set_ui_ui(ray->dcn[1], 0, 0, MPC_RNDNN);
        for (int p = 0; p < m; ++p) {
          for (int w = 0; w < 2; ++w) {
            // dc = 2 * z * dc + 1;
            mpc_mul(ray->dcn[w], ray->zn[w], ray->dcn[w], MPC_RNDNN);
            mpfr_mul_2exp(mpc_realref(ray->dcn[w]), mpc_realref(ray->dcn[w]), 1, ↙
```

```
            ↪ MPFR_RNDN);
          mpfr_mul_2exp(mpc_imagref(ray->dcn[w]), mpc_imagref(ray->dcn[w]), 1, ↙
            ↪ MPFR_RNDN);
          mpfr_add_ui(mpc_realref(ray->dcn[w]), mpc_realref(ray->dcn[w]), 1, ↙
            ↪ MPFR_RNDN);
          // z = z * z + c;
          mpc_sqr(ray->zn[w], ray->zn[w], MPC_RNDNN);
          mpc_add(ray->zn[w], ray->zn[w], ray->cn[w], MPC_RNDNN);
        }
      }
      for (int w = 0; w < 2; ++w) {
        // c_new = c - (z - k) / dc;
        mpc_sub(ray->c_new[w], ray->zn[w], ray->k[w], MPC_RNDNN);
        mpc_div(ray->c_new[w], ray->c_new[w], ray->dcn[w], MPC_RNDNN);
        mpc_sub(ray->c_new[w], ray->cn[w], ray->c_new[w], MPC_RNDNN);
        // c = c_new;
        mpc_set(ray->cn[w], ray->c_new[w], MPC_RNDNN);
        // d2[w] = cabs2(c_new[w] - ray->c);
        mpc_sub(ray->dcn[w], ray->c_new[w], ray->c, MPC_RNDNN);
        mpc_norm(ray->d2[w], ray->dcn[w], MPFR_RNDN);
      }
    /*
      if (! (e2[0] > epsilon2 && e2[1] > epsilon2)) {
        break;
      }
    */
    }
    if ((mpc_finite_p(ray->cn[0]) && mpc_finite_p(ray->cn[1]) && ↙
      ↪ mpfr_lessequal_p(ray->d2[0], ray->d2[1]))
     || (mpc_finite_p(ray->cn[0]) && ! mpc_finite_p(ray->cn[1]))) {
      ray->bit = 0;
      mpc_set(ray->c, ray->cn[0], MPC_RNDNN);
      mpc_set(ray->z, ray->zn[0], MPC_RNDNN);
      mpc_norm(ray->z2, ray->z, MPFR_RNDN);
      ray->n = m;
      ray->d = dwell(ray->loger2, m, mpfr_get_d(ray->z2, MPFR_RNDN));
      return m_stepped;
    }
    if ((mpc_finite_p(ray->cn[0]) && mpc_finite_p(ray->cn[1]) && ↙
      ↪ mpfr_lessequal_p(ray->d2[1], ray->d2[0]))
     || (! mpc_finite_p(ray->cn[0]) && mpc_finite_p(ray->cn[1]))) {
      ray->bit = 1;
      mpc_set(ray->c, ray->cn[1], MPC_RNDNN);
      mpc_set(ray->z, ray->zn[1], MPC_RNDNN);
      mpc_norm(ray->z2, ray->z, MPFR_RNDN);
      ray->n = m;
      ray->d = dwell(ray->loger2, m, mpfr_get_d(ray->z2, MPFR_RNDN));
      return m_stepped;
    }
    return m_failed;
  }
}

extern bool m_r_exray_out_have_bit(const m_r_exray_out * ray) {
  if (! ray) {
    return false;
  }
```

```
  return 0 <= ray->bit;
}

extern bool m_r_exray_out_get_bit(const m_r_exray_out *ray) {
  if (! ray) {
    return false;
  }
  return ray->bit;
}

extern void m_r_exray_out_get(const m_r_exray_out *ray, mpc_t c) {
  if (! ray) {
    return;
  }
  mpfr_prec_t precr = mpfr_get_prec(mpc_realref(ray->c));
  mpfr_prec_t preci = mpfr_get_prec(mpc_imagref(ray->c));
  mpfr_prec_t prec = precr > preci ? precr : preci;
  mpfr_set_prec(mpc_realref(c), prec);
  mpfr_set_prec(mpc_imagref(c), prec);
  mpc_set(c, ray->c, MPC_RNDNN);
}

extern char *m_r_exray_out_do(const mpc_t c, int sharpness, int maxdwell, double er) {
  m_r_exray_out *ray = m_r_exray_out_new(c, sharpness, maxdwell, er);
  if (! ray) {
    return 0;
  }
  char *bits = malloc(maxdwell + 2);
  if (! bits) {
    m_r_exray_out_delete(ray);
    return 0;
  }
  int n = 0;
  while (n <= maxdwell) {
    if (m_r_exray_out_have_bit(ray)) {
      bits[n++] = '0' + m_r_exray_out_get_bit(ray);
    }
    if (m_stepped != m_r_exray_out_step(ray)) {
      break;
    }
  }
  bits[n] = 0;
  m_r_exray_out_delete(ray);
  return bits;
}
```

68 c/lib/m_r_exray_out_perturbed.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2019 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

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

static inline bool mpc_finite_p(const mpc_t z) {
```

```
  return mpfr_number_p(mpc_realref(z)) && mpfr_number_p(mpc_imagref(z));
}

static double dwell(double loger2, int n, double zmag2) {
  return n - log2(log(zmag2) / loger2);
}

struct m_r_exray_out_perturbed {
  int period;
  double _Complex *orbit;
  int sharpness;
  double d;
  double er;
  double er2;
  double loger2;
  double _Complex delta_c;
  double _Complex z;
  int n;
  int bit;
};

extern m_r_exray_out_perturbed *m_r_exray_out_perturbed_new(const mpc_t nucleus,
    ↪ int period, const mpc_t endpoint, int sharpness, int maxdwell, double er)
    ↪ {
  m_r_exray_out_perturbed *ray = malloc(sizeof(*ray));
  if (! ray) {
    return 0;
  }
  ray->period = period;
  ray->orbit = calloc(1, sizeof(ray->orbit[0]) * period);
  if (! ray->orbit)
  {
    free(ray);
    return 0;
  }
  mpfr_prec_t precr = mpfr_get_prec(mpc_realref(nucleus));
  mpfr_prec_t preci = mpfr_get_prec(mpc_imagref(nucleus));
  mpfr_prec_t prec = precr > preci ? precr : preci;
  mpc_t Z;
  mpc_init2(Z, prec);
  mpc_set_dc(Z, 0, MPC_RNDNN);
  for (int i = 0; i < period; ++i)
  {
    ray->orbit[i] = mpc_get_dc(Z, MPC_RNDNN);
    mpc_sqr(Z, Z, MPC_RNDNN);
    mpc_add(Z, Z, nucleus, MPC_RNDNN);
  }
  mpc_sub(Z, endpoint, nucleus, MPC_RNDNN);
  ray->delta_c = mpc_get_dc(Z, MPC_RNDNN);
  mpc_clear(Z);
  double er2 = er * er;
  ray->er = er;
  ray->er2 = er2;
  ray->loger2 = log(er2);
  int n = 0;
  double _Complex z = 0;
  for (int i = 0; i < maxdwell; ++i) {
```

```
      z = (2 * ray->orbit[i % ray->period] + z) * z + ray->delta_c;
      n = n + 1;
      if (cabs2(ray->orbit[n % ray->period] + z) > ray->er2) {
        break;
      }
    }
    if (! (cabs2(ray->orbit[n % ray->period] + z) > ray->er2)) {
      // FIXME cleanup
      return 0;
    }
    ray->sharpness = sharpness;
    ray->z = ray->orbit[n % ray->period] + z;
    ray->d = dwell(ray->loger2, n, cabs2(ray->z));
    ray->n = n;
    ray->bit = -1;
    return ray;
}

extern void m_r_exray_out_perturbed_delete(m_r_exray_out_perturbed *ray) {
    if (ray) {
      if (ray->orbit) {
        free(ray->orbit);
      }
      free(ray);
    }
}

extern m_newton m_r_exray_out_perturbed_step(m_r_exray_out_perturbed *ray) {
    if (! ray) {
      return m_failed;
    }
    ray->bit = -1;
    ray->d -= 1.0 / ray->sharpness;
    if (ray->d <= 0) {
      return m_converged;
    }
    int m = ceil(ray->d);
    double r = pow(ray->er, pow(2, m - ray->d));
    double a = carg(ray->z) / twopi;
    double t = a - floor(a);

    if (m == ray->n) {
      double _Complex k = r * cexp(I * twopi *  t);
      double _Complex c = ray->delta_c;
      double _Complex Z, z, dc;
      for (int i = 0; i < 8; ++i) { // FIXME arbitrary limit
        z = 0;
        dc = 0;
        for (int p = 0; p < m; ++p) {
          Z = ray->orbit[p % ray->period];
          dc = 2 * (Z + z) * dc + 1;
          z = (2 * Z + z) * z + c;
        }
        Z = ray->orbit[m % ray->period];
        c -= (Z + z - k) / dc;
      }
      if (cisfinite(c))
```

```
      {
        ray->delta_c = c;
        ray->z = Z + z;
        ray->n = m;
        ray->d = dwell(ray->loger2, m, cabs2(ray->z));
        return m_stepped;
      }
      return m_failed;

    } else {
      double _Complex k[2] =
        { r * cexp(I * pi *  t      )
        , r * cexp(I * pi * (t + 1))
        };
      double _Complex c[2] = { ray->delta_c, ray->delta_c };
      double _Complex z[2], Z;
      double d2[2];
      for (int i = 0; i < 8; ++i) { // FIXME arbitrary limit
        z[0] = 0;
        z[1] = 0;
        double _Complex dc[2] = { 0, 0 };
        for (int p = 0; p < m; ++p) {
          Z = ray->orbit[p % ray->period];
          for (int w = 0; w < 2; ++w) {
            dc[w] = 2 * (Z + z[w]) * dc[w] + 1;
            z[w] = (2 * Z + z[w]) * z[w] + c[w];
          }
        }
        Z = ray->orbit[m % ray->period];
        for (int w = 0; w < 2; ++w) {
          c[w] -= (Z + z[w] - k[w]) / dc[w];
          d2[w] = cabs(c[w] - ray->delta_c);
        }
      }
      if ((cisfinite(c[0]) && cisfinite(c[1]) && d2[0] <= d2[1]) || (cisfinite(c↙
          ↳ [0]) && ! cisfinite(c[1]))) {
        ray->bit = 0;
        ray->delta_c = c[0];
        ray->z = Z + z[0];
        ray->n = m;
        ray->d = dwell(ray->loger2, m, cabs2(ray->z));
        return m_stepped;
      }
      if ((cisfinite(c[0]) && cisfinite(c[1]) && d2[1] <= d2[0]) || (cisfinite(c↙
          ↳ [1]) && ! cisfinite(c[0]))) {
        ray->bit = 1;
        ray->delta_c = c[1];
        ray->z = Z + z[1];
        ray->n = m;
        ray->d = dwell(ray->loger2, m, cabs2(ray->z));
        return m_stepped;
      }
      return m_failed;
    }
  }

  extern bool m_r_exray_out_perturbed_have_bit(const m_r_exray_out_perturbed *ray)↙
```

```
    ↪ {
  if (! ray) {
    return false;
  }
  return 0 <= ray->bit;
}

extern bool m_r_exray_out_perturbed_get_bit(const m_r_exray_out_perturbed *ray) ↙
    ↪ {
  if (! ray) {
    return false;
  }
  return ray->bit;
}

extern double _Complex m_r_exray_out_perturbed_get(const m_r_exray_out_perturbed↙
    ↪ *ray) {
  if (! ray) {
    return 0;
  }
  return ray->delta_c;
}

extern char *m_r_exray_out_perturbed_do(const mpc_t nucleus, int period, const ↙
    ↪ mpc_t c, int sharpness, int maxdwell, double er) {
  m_r_exray_out_perturbed *ray = m_r_exray_out_perturbed_new(nucleus, period, c,↙
      ↪ sharpness, maxdwell, er);
  if (! ray) {
    return 0;
  }
  char *bits = malloc(maxdwell + 2);
  if (! bits) {
    m_r_exray_out_perturbed_delete(ray);
    return 0;
  }
  int n = 0;
  while (n <= maxdwell) {
    if (m_r_exray_out_perturbed_have_bit(ray)) {
      bits[n++] = '0' + m_r_exray_out_perturbed_get_bit(ray);
    }
    if (m_stepped != m_r_exray_out_perturbed_step(ray)) {
      break;
    }
  }
  bits[n] = 0;
  m_r_exray_out_perturbed_delete(ray);
  return bits;
}

extern char *m_r_exray_out_perturbed_do_r(const mpfr_t nre, const mpfr_t nim, ↙
    ↪ int period, const mpfr_t ere, const mpfr_t eim, int sharpness, int ↙
    ↪ maxdwell, double er)
{
  mpc_t n, e;
  mpfr_prec_t precr = mpfr_get_prec(nre);
  mpfr_prec_t preci = mpfr_get_prec(nim);
  mpfr_prec_t prec = precr > preci ? precr : preci;
```

```
  mpc_init2(n, prec);
  mpc_init2(e, prec);
  mpfr_set(mpc_realref(n), nre, MPFR_RNDN);
  mpfr_set(mpc_imagref(n), nim, MPFR_RNDN);
  mpfr_set(mpc_realref(e), ere, MPFR_RNDN);
  mpfr_set(mpc_imagref(e), eim, MPFR_RNDN);
  char *r = m_r_exray_out_perturbed_do(n, period, e, sharpness, maxdwell, er);
  mpc_clear(n);
  mpc_clear(e);
  return r;
}
```

69 c/lib/m_r_external_angles.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

// for strdup()
#define _POSIX_C_SOURCE 200809L

#include <complex.h>
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <mandelbrot-numerics.h>
#include "m_d_util.h"

extern void m_r_external_angles(char **lo, char **hi, const mpc_t nucleus, int ↙
    ↳ period, double radius, int resolution)
{
  double er = 32;
  if ((! lo) || (! hi))
  {
    return;
  }
  if (period <= 1)
  {
    *lo = strdup(".(0)");
    *hi = strdup(".(1)");
    return;
  }
  mpfr_prec_t precr = mpfr_get_prec(mpc_realref(nucleus));
  mpfr_prec_t preci = mpfr_get_prec(mpc_imagref(nucleus));
  mpfr_prec_t prec = precr > preci ? precr : preci;
  mpfr_t r;
  mpfr_init2(r, 53);
  m_r_domain_size(r, nucleus, period);
  mpfr_mul_d(r, r, radius, MPFR_RNDN);
  int *counts = malloc(resolution * sizeof(int));
  int maxiter = 16 * period;
  mpc_t probe, z;
  mpc_init2(probe, prec);
  mpc_init2(z, prec);
  mpfr_t mz, er2;
  mpfr_init2(mz, 53);
```

```
  mpfr_init2(er2, 53);
  mpfr_set_d(er2, er * er, MPFR_RNDN);
  for (int t = 0; t < resolution; ++t)
  {
    double a = 2 * pi * (t + 0.5) / resolution;
    double c = cos(a);
    double s = sin(a);
    // probe = nucleus + r * (c + I * s);
    mpc_set_d_d(probe, c, s, MPC_RNDNN);
    mpc_mul_fr(probe, probe, r, MPC_RNDNN);
    mpc_add(probe, probe, nucleus, MPC_RNDNN);
    // z = 0
    mpc_set_d(z, 0, MPC_RNDNN);
    int count = maxiter;
    for (int i = 0; i < maxiter; ++i)
    {
      // z = z * z + probe;
      mpc_sqr(z, z, MPC_RNDNN);
      mpc_add(z, z, probe, MPC_RNDNN);
      // mz = cabs2(z);
      mpc_norm(mz, z, MPFR_RNDN);
      if (mpfr_greater_p(mz, er2))
      {
        count = i;
        break;
      }
    }
    counts[t] = count;
  }
  char *first = 0;
  while (1)
  {
    int mint = -1;
    int mincount = maxiter;
    for (int t = 0; t < resolution; ++t)
    {
      if (counts[t] < mincount)
      {
        mincount = counts[t];
        mint = t;
      }
    }
    if (mint == -1)
    {
      if (first)
        free(first);
      *lo = 0;
      *hi = 0;
      return;
    }
    double a = 2 * pi * (mint + 0.5) / resolution;
    double c = cos(a);
    double s = sin(a);
    // probe = nucleus + r * (c + I * s);
    mpc_set_d_d(probe, c, s, MPC_RNDNN);
    mpc_mul_fr(probe, probe, r, MPC_RNDNN);
    mpc_add(probe, probe, nucleus, MPC_RNDNN);
```

```
100       char *bits = m_r_exray_out_do(probe, 8, mincount * 2, er);
          if (bits)
          {
            int bitlen = strlen(bits);
            if (bitlen >= period)
105         {
              char *angle = malloc(period + 4);
              angle[0] = '.';
              angle[1] = '(';
              for (int i = 0; i < period; ++i)
110           {
                angle[2 + i] = bits[bitlen-1 - i];
              }
              angle[2 + period] = ')';
              angle[3 + period] = 0;
115           free(bits);
              if (first)
              {
                if (strcmp(first, angle) != 0)
                {
120               if (strcmp(first, angle) < 0)
                  {
                    *lo = first;
                    *hi = angle;
                  }
125               else
                  {
                    *lo = angle;
                    *hi = first;
                  }
130               free(counts);
                  return;
                }
                else
                {
135               free(angle);
                }
              }
              else
              {
140             first = angle;
                angle = 0;
              }
            }
          }
145       counts[mint] = maxiter;
        }
      }
```

70 c/lib/m_r_interior.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

5 #include <mandelbrot-numerics.h>
```

```
extern m_newton m_r_interior_step_raw(mpc_t z_out, mpc_t c_out, const mpc_t z_guess, const mpc_t c_guess, const mpc_t interior, int period, mpc_t z_new, mpc_t c_new, mpc_t c, mpc_t z, mpc_t dz, mpc_t dc, mpc_t dzdz, mpc_t dcdz, mpc_t det, mpc_t d, mpc_t dz1, mpfr_t d2z, mpfr_t d2c, mpfr_t epsilon2) {
  // c = c_guess; z = z_guess; dz = 1; dc = 0; dzdz = 0; dcdz = 0;
  mpc_set(c, c_guess, MPC_RNDNN);
  mpc_set(z, z_guess, MPC_RNDNN);
  mpc_set_si(dz, 1, MPC_RNDNN);
  mpc_set_si(dc, 0, MPC_RNDNN);
  mpc_set_si(dzdz, 0, MPC_RNDNN);
  mpc_set_si(dcdz, 0, MPC_RNDNN);
  for (int p = 0; p < period; ++p) {
    // dcdz = 2 * (z * dcdz + dc * dz);
    mpc_mul(dcdz, z, dcdz, MPC_RNDNN);
    mpc_mul(d, dc, dz, MPC_RNDNN);
    mpc_add(dcdz, dcdz, d, MPC_RNDNN);
    mpc_mul_2si(dcdz, dcdz, 1, MPC_RNDNN);
    // dzdz = 2 * (z * dzdz + dz * dz);
    mpc_mul(dzdz, z, dzdz, MPC_RNDNN);
    mpc_sqr(d, dz, MPC_RNDNN);
    mpc_add(dzdz, dzdz, d, MPC_RNDNN);
    mpc_mul_2si(dzdz, dzdz, 1, MPC_RNDNN);
    // dc = 2 * z * dc + 1;
    mpc_mul(dc, z, dc, MPC_RNDNN);
    mpc_mul_2si(dc, dc, 1, MPC_RNDNN);
    mpc_add_ui(dc, dc, 1, MPC_RNDNN);
    // dz = 2 * z * dz;
    mpc_mul(dz, z, dz, MPC_RNDNN);
    mpc_mul_2si(dz, dz, 1, MPC_RNDNN);
    // z = z * z + c;
    mpc_sqr(z, z, MPC_RNDNN);
    mpc_add(z, z, c, MPC_RNDNN);
  }
  // det = (dz - 1) * dcdz - dc * dzdz;
  mpc_sub_ui(dz1, dz, 1, MPC_RNDNN);
  mpc_mul(det, dz1, dcdz, MPC_RNDNN);
  mpc_mul(d, dc, dzdz, MPC_RNDNN);
  mpc_sub(det, det, d, MPC_RNDNN);
  // z_new = z_guess - (dcdz * (z - z_guess) - dc * (dz - interior)) / det;
  mpc_sub(z, z, z_guess, MPC_RNDNN);
  mpc_sub(dz, dz, interior, MPC_RNDNN);
  mpc_mul(dcdz, dcdz, z, MPC_RNDNN);
  mpc_mul(dc, dc, dz, MPC_RNDNN);
  mpc_sub(dcdz, dcdz, dc, MPC_RNDNN);
  mpc_div(dcdz, dcdz, det, MPC_RNDNN);
  mpc_sub(z_new, z_guess, dcdz, MPC_RNDNN);
  // c_new = c_guess - ((dz - 1) * (dz - interior) - dzdz * (z - z_guess)) / det;
  mpc_mul(dz1, dz1, dz, MPC_RNDNN);
  mpc_mul(dzdz, dzdz, z, MPC_RNDNN);
  mpc_sub(dzdz, dz1, dzdz, MPC_RNDNN);
  mpc_div(dzdz, dzdz, det, MPC_RNDNN);
  mpc_sub(c_new, c_guess, dzdz, MPC_RNDNN);
  if (mpfr_number_p(mpc_realref(z_new)) && mpfr_number_p(mpc_imagref(z_new)) &&
      mpfr_number_p(mpc_realref(c_new)) && mpfr_number_p(mpc_imagref(c_new))) {
    // z_out = z_new; d2z = norm(z_new - z_guess);
```

```
    mpc_sub(d, z_new, z_guess, MPC_RNDNN); // in this order in case z_out = z_guess
    mpc_set(z_out, z_new, MPC_RNDNN);
    mpc_norm(d2z, d, MPFR_RNDN);
    // c_out = c_new; d2c = norm(c_new - c_guess);
    mpc_sub(d, c_new, c_guess, MPC_RNDNN); // in this order in case c_out = c_guess
    mpc_set(c_out, c_new, MPC_RNDNN);
    mpc_norm(d2c, d, MPFR_RNDN);
    if (mpfr_lessequal_p(d2z, epsilon2) && mpfr_lessequal_p(d2c, epsilon2)) {
      return m_converged;
    } else {
      return m_stepped;
    }
  } else {
    // z_out = z_guess; c_out = c_guess;
    mpc_set(z_out, z_guess, MPC_RNDNN);
    mpc_set(c_out, c_guess, MPC_RNDNN);
    return m_failed;
  }
}

extern m_newton m_r_interior_step(mpc_t z_out, mpc_t c_out, const mpc_t z_guess, const mpc_t c_guess, const mpc_t interior, int period) {
  m_newton result = m_failed;
  // prec
  mpfr_prec_t precr, preci, prec;
  mpc_get_prec2(&precr, &preci, z_guess);
  prec = precr > preci ? precr : preci;
  mpc_get_prec2(&precr, &preci, c_guess);
  prec = precr > prec ? precr : prec;
  prec = preci > prec ? preci : prec;
  mpc_set_prec(z_out, prec); // FIXME might trash when z_out = z_guess
  mpc_set_prec(c_out, prec); // FIXME might trash when c_out = c_guess
  // init
  mpc_t z_new, c_new, c, z, dz, dc, dzdz, dcdz, det, d, dz1;
  mpfr_t d2z, d2c, epsilon2;
  mpc_init2(z_new, prec);
  mpc_init2(c_new, prec);
  mpc_init2(c, prec);
  mpc_init2(z, prec);
  mpc_init2(dc, prec);
  mpc_init2(dz, prec);
  mpc_init2(dcdz, prec);
  mpc_init2(dzdz, prec);
  mpc_init2(det, prec);
  mpc_init2(d, prec);
  mpc_init2(dz1, prec);
  mpfr_init2(d2z, prec);
  mpfr_init2(d2c, prec);
  mpfr_init2(epsilon2, prec);
  // epsilon
  mpfr_set_si(epsilon2, 2, MPFR_RNDN);
  mpfr_nextabove(epsilon2);
  mpfr_sub_si(epsilon2, epsilon2, 2, MPFR_RNDN);
  mpfr_sqr(epsilon2, epsilon2, MPFR_RNDN);
  // step
```

```
  result = m_r_interior_step_raw(z_out, c_out, z_guess, c_guess, interior, ↙
      ↳ period, z_new, c_new, c, z, dz, dc, dzdz, dcdz, det, d, dz1, d2z, d2c, ↙
      ↳ epsilon2);
  // cleanup
  mpc_clear(z_new);
  mpc_clear(c_new);
  mpc_clear(c);
  mpc_clear(z);
  mpc_clear(dc);
  mpc_clear(dz);
  mpc_clear(dcdz);
  mpc_clear(dzdz);
  mpc_clear(det);
  mpc_clear(d);
  mpc_clear(dz1);
  mpfr_clear(d2z);
  mpfr_clear(d2c);
  mpfr_clear(epsilon2);
  return result;
}

extern m_newton m_r_interior(mpc_t z_out, mpc_t c_out, const mpc_t z_guess, ↙
    ↳ const mpc_t c_guess, const mpc_t interior, int period, int maxsteps) {
  m_newton result = m_failed;
  // prec
  mpfr_prec_t precr, preci, prec;
  mpc_get_prec2(&precr, &preci, z_guess);
  prec = precr > preci ? precr : preci;
  mpc_get_prec2(&precr, &preci, c_guess);
  prec = precr > prec ? precr : prec;
  prec = preci > prec ? preci : prec;
  // init
  mpc_t z, c, z_new, c_new, cc, zz, dz, dc, dzdz, dcdz, det, d, dz1;
  mpfr_t d2z, d2c, epsilon2;
  mpc_init2(z, prec);
  mpc_init2(c, prec);
  mpc_init2(z_new, prec);
  mpc_init2(c_new, prec);
  mpc_init2(cc, prec);
  mpc_init2(zz, prec);
  mpc_init2(dc, prec);
  mpc_init2(dz, prec);
  mpc_init2(dcdz, prec);
  mpc_init2(dzdz, prec);
  mpc_init2(det, prec);
  mpc_init2(d, prec);
  mpc_init2(dz1, prec);
  mpfr_init2(d2z, prec);
  mpfr_init2(d2c, prec);
  mpfr_init2(epsilon2, prec);
  // epsilon
  mpfr_set_si(epsilon2, 2, MPFR_RNDN);
  mpfr_nextabove(epsilon2);
  mpfr_sub_si(epsilon2, epsilon2, 2, MPFR_RNDN);
  mpfr_sqr(epsilon2, epsilon2, MPFR_RNDN);
  // z = z_guess; c = c_guess;
  mpc_set(z, z_guess, MPC_RNDNN);
```

```
  mpc_set(c, c_guess, MPC_RNDNN);
  for (int i = 0; i < maxsteps; ++i) {
    if (m_stepped != (result = m_r_interior_step_raw(z, c, z, c, interior, ↙
        ↳ period, z_new, c_new, cc, zz, dz, dc, dzdz, dcdz, det, d, dz1, d2z, ↙
        ↳ d2c, epsilon2))) {
      break;
    }
  }
  // z_out = z; c_out = c;
  mpc_set_prec(z_out, prec);
  mpc_set(z_out, z, MPC_RNDNN);
  mpc_set_prec(c_out, prec);
  mpc_set(c_out, c, MPC_RNDNN);
  // cleanup
  mpc_clear(z);
  mpc_clear(c);
  mpc_clear(z_new);
  mpc_clear(c_new);
  mpc_clear(cc);
  mpc_clear(zz);
  mpc_clear(dc);
  mpc_clear(dz);
  mpc_clear(dcdz);
  mpc_clear(dzdz);
  mpc_clear(det);
  mpc_clear(d);
  mpc_clear(dz1);
  mpfr_clear(d2z);
  mpfr_clear(d2c);
  mpfr_clear(epsilon2);
  return result;
}
```

71 c/lib/m_r_misiurewicz.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

#include <mandelbrot-numerics.h>

extern m_newton m_r_misiurewicz_naive_step_raw(mpc_t c_out, const mpc_t c_guess, ↙
    ↳  int preperiod, int period, mpc_t z, mpc_t dc, mpc_t zp, mpc_t dcp, mpc_t ↙
    ↳ c_new, mpc_t d, mpfr_t d2, mpfr_t epsilon2) {
  mpc_set_ui_ui(z, 0, 0, MPC_RNDNN);
  mpc_set_ui_ui(dc, 0, 0, MPC_RNDNN);
  mpc_set_ui_ui(zp, 0, 0, MPC_RNDNN);
  mpc_set_ui_ui(dcp, 0, 0, MPC_RNDNN);
  for (int i = 0; i < preperiod + period; ++i) {
    if (i == preperiod) {
      mpc_set(zp, z, MPC_RNDNN);
      mpc_set(dcp, dc, MPC_RNDNN);
    }
    // dc = 2 * z * dc + 1;
    mpc_mul(dc, z, dc, MPC_RNDNN);
    mpc_mul_2si(dc, dc, 1, MPC_RNDNN);
    mpc_add_ui(dc, dc, 1, MPC_RNDNN);
```

```
      // z = z * z + c_guess;
      mpc_sqr(z, z, MPC_RNDNN);
      mpc_add(z, z, c_guess, MPC_RNDNN);
    }
    mpc_sub(z, z, zp, MPC_RNDNN);
    mpc_sub(dc, dc, dcp, MPC_RNDNN);
    mpc_div(zp, z, dc, MPC_RNDNN);
    mpc_sub(c_new, c_guess, zp, MPC_RNDNN);
    mpc_sub(d, c_new, c_guess, MPC_RNDNN);
    mpc_norm(d2, d, MPFR_RNDN);
    if (mpfr_less_p(d2, epsilon2)) {
      mpc_set(c_out, c_new, MPC_RNDNN);
      return m_converged;
    }
    if (mpfr_number_p(mpc_realref(d)) && mpfr_number_p(mpc_imagref(d))) {
      mpc_set(c_out, c_new, MPC_RNDNN);
      return m_stepped;
    } else {
      mpc_set(c_out, c_guess, MPC_RNDNN);
      return m_failed;
    }
  }

  extern m_newton m_r_misiurewicz_naive_step(mpc_t c_out, const mpc_t c_guess, int preperiod, int period) {
    // prec
    mpfr_prec_t precr, preci, prec;
    mpc_get_prec2(&precr, &preci, c_guess);
    prec = precr > preci ? precr : preci;
    mpc_set_prec(c_out, prec); // FIXME might trash when c_out = c_guess
    // init
    mpc_t z, dc, zp, dcp, c_new, d;
    mpfr_t d2, epsilon2;
    mpc_init2(z, prec);
    mpc_init2(dc, prec);
    mpc_init2(zp, prec);
    mpc_init2(dcp, prec);
    mpc_init2(c_new, prec);
    mpc_init2(d, prec);
    mpfr_init2(d2, prec);
    mpfr_init2(epsilon2, prec);
    // epsilon
    mpfr_set_si(epsilon2, 2, MPFR_RNDN);
    mpfr_nextabove(epsilon2);
    mpfr_sub_si(epsilon2, epsilon2, 2, MPFR_RNDN);
    mpfr_sqr(epsilon2, epsilon2, MPFR_RNDN);
    // step raw
    m_newton retval = m_r_misiurewicz_naive_step_raw(c_out, c_guess, preperiod, period, z, dc, zp, dcp, c_new, d, d2, epsilon2);
    // cleanup
    mpc_clear(z);
    mpc_clear(dc);
    mpc_clear(zp);
    mpc_clear(dcp);
    mpc_clear(c_new);
    mpc_clear(d);
    mpfr_clear(d2);
```

```
  mpfr_clear(epsilon2);
  return retval;
}

extern m_newton m_r_misiurewicz_naive(mpc_t c_out, const mpc_t c_guess, int ↙
    ↳ preperiod, int period, int maxsteps) {
  // prec
  mpfr_prec_t precr, preci, prec;
  mpc_get_prec2(&precr, &preci, c_guess);
  prec = precr > preci ? precr : preci;
  // init
  mpc_t c, z, dc, zp, dcp, c_new, d;
  mpfr_t d2, epsilon2;
  mpc_init2(c, prec);
  mpc_init2(z, prec);
  mpc_init2(dc, prec);
  mpc_init2(zp, prec);
  mpc_init2(dcp, prec);
  mpc_init2(c_new, prec);
  mpc_init2(d, prec);
  mpfr_init2(d2, prec);
  mpfr_init2(epsilon2, prec);
  // epsilon
  mpfr_set_si(epsilon2, 2, MPFR_RNDN);
  mpfr_nextabove(epsilon2);
  mpfr_sub_si(epsilon2, epsilon2, 2, MPFR_RNDN);
  mpfr_sqr(epsilon2, epsilon2, MPFR_RNDN);
  // c = c_guess
  m_newton retval = m_failed;
  mpc_set(c, c_guess, MPC_RNDNN);
  for (int i = 0; i < maxsteps; ++i) {
    if (m_stepped != (retval = m_r_misiurewicz_naive_step_raw(c, c, preperiod, ↙
        ↳ period, z, dc, zp, dcp, c_new, d, d2, epsilon2))) {
      break;
    }
  }
  // c_out = c;
  mpc_set_prec(c_out, prec);
  mpc_set(c_out, c, MPC_RNDNN);
  // cleanup
  mpc_clear(c);
  mpc_clear(z);
  mpc_clear(dc);
  mpc_clear(zp);
  mpc_clear(dcp);
  mpc_clear(c_new);
  mpc_clear(d);
  mpfr_clear(d2);
  mpfr_clear(epsilon2);
  return retval;
}

extern m_newton m_r_misiurewicz_step_raw(mpc_t c_out, const mpc_t c_guess, int ↙
    ↳ preperiod, int period, mpc_t z, mpc_t dc, mpc_t zp, mpc_t dcp, mpc_t f, ↙
    ↳ mpc_t df, mpc_t g, mpc_t dg, mpc_t h, mpc_t dh, mpc_t k, mpc_t l, mpfr_t ↙
    ↳ epsilon2) {
  // iteration
```

```
    // z = 0; dc = 0; zp = 0; dcp = 0; h = 1; dh = 0;
    mpc_set_si(z, 0, MPC_RNDNN);
    mpc_set_si(dc, 0, MPC_RNDNN);
    mpc_set_si(zp, 0, MPC_RNDNN);
    mpc_set_si(dcp, 0, MPC_RNDNN);
    mpc_set_si(h, 1, MPC_RNDNN);
    mpc_set_si(dh, 0, MPC_RNDNN);
    for (int i = 0; i < period; ++i) {
      // dc = 2 * z * dc + 1;
      mpc_mul(dc, z, dc, MPC_RNDNN);
      mpc_mul_2si(dc, dc, 1, MPC_RNDNN);
      mpc_add_ui(dc, dc, 1, MPC_RNDNN);
      // z = z * z + c_guess;
      mpc_sqr(z, z, MPC_RNDNN);
      mpc_add(z, z, c_guess, MPC_RNDNN);
    }
    for (int i = 0; i < preperiod; ++i) {
      // reject lower preperiods
      // k = z - zp;
      mpc_sub(k, z, zp, MPC_RNDNN);
      // h = h * k;
      mpc_mul(h, h, k, MPC_RNDNN);
      // dh = dh + (dc - dcp) / k;
      mpc_sub(l, dc, dcp, MPC_RNDNN);
      mpc_div(l, l, k, MPC_RNDNN);
      mpc_add(dh, dh, l, MPC_RNDNN);
      // iterate
      // dc = 2 * z * dc + 1;
      mpc_mul(dc, z, dc, MPC_RNDNN);
      mpc_mul_2si(dc, dc, 1, MPC_RNDNN);
      mpc_add_ui(dc, dc, 1, MPC_RNDNN);
      // z = z * z + c_guess;
      mpc_sqr(z, z, MPC_RNDNN);
      mpc_add(z, z, c_guess, MPC_RNDNN);
      // dcp = 2 * zp * dcp + 1;
      mpc_mul(dcp, zp, dcp, MPC_RNDNN);
      mpc_mul_2si(dcp, dcp, 1, MPC_RNDNN);
      mpc_add_ui(dcp, dcp, 1, MPC_RNDNN);
      // zp = zp * zp + c_guess;
      mpc_sqr(zp, zp, MPC_RNDNN);
      mpc_add(zp, zp, c_guess, MPC_RNDNN);
    }
    // build function
    // dh = dh * h;
    mpc_mul(dh, dh, h, MPC_RNDNN);
    // g = z - zp;
    mpc_sub(g, z, zp, MPC_RNDNN);
    // dg = dc - dcp;
    mpc_sub(dg, dc, dcp, MPC_RNDNN);
    // f = g / h;
    mpc_div(f, g, h, MPC_RNDNN);
    // df = (dg * h - g * dh) / (h * h);
    mpc_mul(dg, dg, h, MPC_RNDNN);
    mpc_mul(dh, dh, g, MPC_RNDNN);
    mpc_sub(df, dg, dh, MPC_RNDNN);
    mpc_sqr(h, h, MPC_RNDNN);
    mpc_div(df, df, h, MPC_RNDNN);
```

```
185   // newton step
      // h = c_guess - f / df;
      mpc_div(g, f, df, MPC_RNDNN);
      mpc_sub(h, c_guess, g, MPC_RNDNN);
      // check convergence
190   // g = h - c_guess;
      mpc_sub(g, h, c_guess, MPC_RNDNN);
      mpc_norm(mpc_realref(f), g, MPFR_RNDN);
      if (mpfr_less_p(mpc_realref(f), epsilon2)) {
        // *c_out = h;
195     mpc_set(c_out, h, MPC_RNDNN);
        return m_converged;
      }
      if (mpfr_number_p(mpc_realref(f))) {
        // *c_out = h;
200     mpc_set(c_out, h, MPC_RNDNN);
        return m_stepped;
      } else {
        // *c_out = c_guess;
        mpc_set(c_out, c_guess, MPC_RNDNN);
205     return m_failed;
      }
    }

    extern m_newton m_r_misiurewicz(mpc_t c_out, const mpc_t c_guess, int preperiod,↙
        ↪ int period, int maxsteps) {
210   // prec
      mpfr_prec_t precr, preci, prec;
      mpc_get_prec2(&precr, &preci, c_guess);
      prec = precr > preci ? precr : preci;
      // init
215   mpc_t c, z, dc, f, df, g, dg, h, dh, k, l, zp, dcp;
      mpfr_t epsilon2;
      mpc_init2(c, prec);
      mpc_init2(z, prec);
      mpc_init2(dc, prec);
220   mpc_init2(zp, prec);
      mpc_init2(dcp, prec);
      mpc_init2(f, prec);
      mpc_init2(df, prec);
      mpc_init2(g, prec);
225   mpc_init2(dg, prec);
      mpc_init2(h, prec);
      mpc_init2(dh, prec);
      mpc_init2(k, prec);
      mpc_init2(l, prec);
230   mpfr_init2(epsilon2, prec);
      // epsilon
      mpfr_set_si(epsilon2, 2, MPFR_RNDN);
      mpfr_nextabove(epsilon2);
      mpfr_sub_si(epsilon2, epsilon2, 2, MPFR_RNDN);
235   mpfr_sqr(epsilon2, epsilon2, MPFR_RNDN);
      // c = c_guess
      m_newton retval = m_failed;
      mpc_set(c, c_guess, MPC_RNDNN);
      for (int i = 0; i < maxsteps; ++i) {
240     if (m_stepped != (retval = m_r_misiurewicz_step_raw(c, c, preperiod, period,↙
```

```
        ↪ z, dc, zp, dcp, f, df, g, dg, h, dh, k, l, epsilon2))) {
      break;
    }
  }
  // c_out = c;
  mpc_set_prec(c_out, prec);
  mpc_set(c_out, c, MPC_RNDNN);
  // cleanup
  mpc_clear(c);
  mpc_clear(z);
  mpc_clear(dc);
  mpc_clear(zp);
  mpc_clear(dcp);
  mpc_clear(f);
  mpc_clear(df);
  mpc_clear(g);
  mpc_clear(dg);
  mpc_clear(h);
  mpc_clear(dh);
  mpc_clear(k);
  mpc_clear(l);
  mpfr_clear(epsilon2);
  return retval;
}
```

72 c/lib/m_r_nucleus.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

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

//#define THREADING
#undef THREADING
#if (!defined(__STDC_NO_THREADS__)) && (!defined(__STDC_NO_ATOMICS__))
#include <threads.h>
#include <stdatomic.h>
typedef void *thrd_result_t;
#else
#if (!defined(__STDC_NO_ATOMICS__))
#include <sched.h>
#include <pthread.h>
#include <stdatomic.h>
typedef pthread_t thrd_t;
typedef void *thrd_result_t;
#define thrd_success 0
#define thrd_create(T,F,A) pthread_create(T,NULL,F,A)
#define thrd_join(T,R) pthread_join(*T,R)
#define thrd_yield() sched_yield()
#else
#undef THREADING
#endif
#endif

#ifdef THREADING
```

```
// centralized barrier from
// https://www.jlab.org/hpc/papers/hpcasia07.pdf

struct barrier_t
{
  atomic_int release;
  int padding1[63];
  atomic_int counter;
  int padding2[63];
  int num_thread;
  bool yielding;
};

void barrier_init(struct barrier_t *b, int num_thread, bool yielding)
{
  b->release = 0;
  b->counter = num_thread;
  b->num_thread = num_thread;
  b->yielding = yielding;
}

void barrier_wait(struct barrier_t *b)
{
  int flag = b->release;
  int count = --(b->counter);
  if (count == 0)
  {
    b->counter = b->num_thread;
    ++(b->release);
  }
  else
    if (b->yielding)
      while (flag == b->release)
        thrd_yield();
    else
      while (flag == b->release)
        ;
}

struct m_r_nucleus_naive_step_common_t
{
  struct barrier_t barrier;
  mpfr_t cr, ci, zr, zi, dr, di, zrdr, zidi, zidr, zrdi, zrzr, zizi, zrzi, d2, ↙
      ↳ epsilon2;
  mpc_t z, dc, c_new, d;
  int period;
};

void m_r_nucleus_naive_step_common_init(struct m_r_nucleus_naive_step_common_t *↙
    ↳ m, mpfr_prec_t prec, int period, int ncpus)
{
  barrier_init(&m->barrier, 7, ncpus < 7);
  mpfr_inits2(prec, m->cr, m->ci, m->zr, m->zi, m->dr, m->di, m->zrdr, m->zidi, ↙
      ↳ m->zidr, m->zrdi, m->zrzr, m->zizi, m->zrzi, m->d2, m->epsilon2, NULL);
  mpc_init2(m->z, prec);
  mpc_init2(m->dc, prec);
```

```
85    mpc_init2(m->c_new, prec);
      mpc_init2(m->d, prec);
      mpfr_set_si(m->epsilon2, 2, MPFR_RNDN);
      mpfr_nextabove(m->epsilon2);
      mpfr_sub_si(m->epsilon2, m->epsilon2, 2, MPFR_RNDN);
90    mpfr_sqr(m->epsilon2, m->epsilon2, MPFR_RNDN);
     m->period = period;
    }

    void m_r_nucleus_naive_step_common_clear(struct m_r_nucleus_naive_step_common_t ↙
        ↳ *m)
95  {
      mpfr_clears(m->cr, m->ci, m->zr, m->zi, m->dr, m->di, m->zrdr, m->zidi, m->↙
          ↳ zidr, m->zrdi, m->zrzr, m->zizi, m->zrzi, m->d2, m->epsilon2, NULL);
      mpc_clear(m->z);
      mpc_clear(m->dc);
      mpc_clear(m->c_new);
100   mpc_clear(m->d);
    }

    struct m_r_nucleus_naive_step_arg_t
    {
105   struct m_r_nucleus_naive_step_common_t *data;
      int id;
    };

    static thrd_result_t m_r_nucleus_naive_step_thread(void *arg0)
110 {
      struct m_r_nucleus_naive_step_arg_t *arg = arg0;
      struct m_r_nucleus_naive_step_common_t *m = arg->data;
      int t = arg->id;
      int period = m->period;
115   for (int p = 0; p < period; ++p)
      {
        switch (t)
        {
          case 0: mpfr_mul(m->zrdr, m->zr, m->dr, MPFR_RNDN); break;
120       case 1: mpfr_mul(m->zidi, m->zi, m->di, MPFR_RNDN); break;
          case 2: mpfr_mul(m->zidr, m->zi, m->dr, MPFR_RNDN); break;
          case 3: mpfr_mul(m->zrdi, m->zr, m->di, MPFR_RNDN); break;
          case 4: mpfr_sqr(m->zrzr, m->zr,        MPFR_RNDN); break;
          case 5: mpfr_sqr(m->zizi, m->zi,        MPFR_RNDN); break;
125       case 6: mpfr_mul(m->zrzi, m->zi, m->zi, MPFR_RNDN); break;
        }
        barrier_wait(&m->barrier);
        switch (t)
        {
130       case 0:
            mpfr_sub(m->dr, m->zrdr, m->zidi, MPFR_RNDN);
            mpfr_mul_2ui(m->dr, m->dr, 1, MPFR_RNDN);
            mpfr_add_ui(m->dr, m->dr, 1, MPFR_RNDN);
            break;
135       case 1:
            mpfr_add(m->di, m->zidr, m->zrdi, MPFR_RNDN);
            mpfr_mul_2ui(m->di, m->di, 1, MPFR_RNDN);
            break;
          case 2:
```

```
140           mpfr_sub(m->zr, m->zrzr, m->zizi, MPFR_RNDN);
              mpfr_add(m->zr, m->zr, m->cr, MPFR_RNDN);
              break;
            case 3:
              mpfr_mul_2ui(m->zi, m->zrzi, 1, MPFR_RNDN);
145           mpfr_add(m->zi, m->zi, m->ci, MPFR_RNDN);
              break;
          }
          barrier_wait(&m->barrier);
          if ((t == 0) && ((p & 0xFFFF) == 0))
150         fprintf(stderr, "%16d\r", p);
        }
        return 0;
      }

155   static m_newton m_r_nucleus_naive_step_raw_threads(mpc_t c_out, const mpc_t ↙
          ↳ c_guess, struct m_r_nucleus_naive_step_common_t *m)
      {
        mpfr_set(m->cr, mpc_realref(c_guess), MPFR_RNDN);
        mpfr_set(m->ci, mpc_imagref(c_guess), MPFR_RNDN);
        mpfr_set_ui(m->zr, 0, MPFR_RNDN);
160     mpfr_set_ui(m->zi, 0, MPFR_RNDN);
        mpfr_set_ui(m->dr, 0, MPFR_RNDN);
        mpfr_set_ui(m->di, 0, MPFR_RNDN);
        struct m_r_nucleus_naive_step_arg_t arg[7];
        thrd_t threads[7];
165     int status[7];
        thrd_result_t result[7] = { 0, 0, 0, 0, 0, 0, 0 };
        bool ok = true;
        for (int t = 0; t < 7; ++t)
        {
170       arg[t].data = m;
          arg[t].id = t;
          status[t] = thrd_create(&threads[t], m_r_nucleus_naive_step_thread, &arg[t])↙
              ↳ ;
          ok = ok && status[t] == thrd_success;
        }
175     for (int t = 0; t < 7; ++t)
        {
          if (status[t] == thrd_success)
            thrd_join(threads[t], &result[t]);
          ok = ok && !result[t];
180     }
        if (! ok)
        {
          mpc_set(c_out, c_guess, MPC_RNDNN);
          return m_failed;
185     }
        mpfr_set(mpc_realref(m->dc), m->dr, MPFR_RNDN);
        mpfr_set(mpc_imagref(m->dc), m->di, MPFR_RNDN);
        mpc_norm(m->d2, m->dc, MPFR_RNDN);
        if (mpfr_lessequal_p(m->d2, m->epsilon2)) {
190       mpc_set(c_out, c_guess, MPC_RNDNN);
          return m_converged;
        }
        // c_new = c_guess - z / dc;
        mpfr_set(mpc_realref(m->z), m->zr, MPFR_RNDN);
```

```
    mpfr_set(mpc_imagref(m->z), m->zi, MPFR_RNDN);
    mpc_div(m->z, m->z, m->dc, MPC_RNDNN);
    mpc_sub(m->c_new, c_guess, m->z, MPC_RNDNN);
    // d = c_new - c_guess;
    mpc_sub(m->d, m->c_new, c_guess, MPC_RNDNN);
    mpc_norm(m->d2, m->d, MPFR_RNDN);
    if (mpfr_lessequal_p(m->d2, m->epsilon2)) {
      mpc_set(c_out, m->c_new, MPC_RNDNN);
      return m_converged;
    }
    if (mpfr_number_p(mpc_realref(m->d)) && mpfr_number_p(mpc_imagref(m->d))) {
      mpc_set(c_out, m->c_new, MPC_RNDNN);
      return m_stepped;
    } else {
      mpc_set(c_out, c_guess, MPC_RNDNN);
      return m_failed;
    }
  }
  #endif

  extern m_newton m_r_nucleus_naive_step_raw(mpc_t c_out, const mpc_t c_guess, int
      ↪ period, mpc_t z, mpc_t dc, mpc_t c_new, mpc_t d, mpfr_t d2, mpfr_t
      ↪ epsilon2) {
    // z = 0; dc = 0;
    mpc_set_si(z, 0, MPC_RNDNN);
    mpc_set_si(dc, 0, MPC_RNDNN);
    for (int i = 0; i < period; ++i) {
      // dc = 2 * z * dc + 1;
      mpc_mul(dc, z, dc, MPC_RNDNN);
      mpc_mul_2si(dc, dc, 1, MPC_RNDNN);
      mpc_add_ui(dc, dc, 1, MPC_RNDNN);
      // z = z * z + c_guess;
      mpc_sqr(z, z, MPC_RNDNN);
      mpc_add(z, z, c_guess, MPC_RNDNN);
    }
    mpc_norm(d2, dc, MPFR_RNDN);
    if (mpfr_lessequal_p(d2, epsilon2)) {
      mpc_set(c_out, c_guess, MPC_RNDNN);
      return m_converged;
    }
    // c_new = c_guess - z / dc;
    mpc_div(z, z, dc, MPC_RNDNN);
    mpc_sub(c_new, c_guess, z, MPC_RNDNN);
    // d = c_new - c_guess;
    mpc_sub(d, c_new, c_guess, MPC_RNDNN);
    mpc_norm(d2, d, MPFR_RNDN);
    if (mpfr_lessequal_p(d2, epsilon2)) {
      mpc_set(c_out, c_new, MPC_RNDNN);
      return m_converged;
    }
    if (mpfr_number_p(mpc_realref(d)) && mpfr_number_p(mpc_imagref(d))) {
      mpc_set(c_out, c_new, MPC_RNDNN);
      return m_stepped;
    } else {
      mpc_set(c_out, c_guess, MPC_RNDNN);
      return m_failed;
    }
```

```
250 }

extern m_newton m_r_nucleus_naive(mpc_t c_out, const mpc_t c_guess, int period, ↙
    ↳ int maxsteps, int ncpus) {
  m_newton result = m_failed;
  // prec
255   mpfr_prec_t precr, preci, prec;
  mpc_get_prec2(&precr, &preci, c_guess);
  prec = precr > preci ? precr : preci;
  // init
#ifdef THREADING
260   if (ncpus > 1)
  {
    // c = c_guess
    mpc_t c;
    mpc_init2(c, prec);
265     mpc_set(c, c_guess, MPC_RNDNN);
    {
      struct m_r_nucleus_naive_step_common_t common;
      m_r_nucleus_naive_step_common_init(&common, prec, period, ncpus);
      for (int i = 0; i < maxsteps; ++i) {
270         fprintf(stderr, "pass\t%8d\n", i);
        if (m_stepped != (result = m_r_nucleus_naive_step_raw_threads(c, c, &↙
            ↳ common))) {
          break;
        }
      }
275     m_r_nucleus_naive_step_common_clear(&common);
    }
    // c_out = c;
    mpc_set_prec(c_out, prec);
    mpc_set(c_out, c, MPC_RNDNN);
280     mpc_clear(c);
  }
  else
#endif
  {
285     (void) ncpus;
    mpc_t c, z, dc, c_new, d;
    mpfr_t d2, epsilon2;
    mpc_init2(c, prec);
    mpc_init2(z, prec);
290     mpc_init2(dc, prec);
    mpc_init2(c_new, prec);
    mpc_init2(d, prec);
    mpfr_init2(d2, prec);
    mpfr_init2(epsilon2, prec);
295     // epsilon
    mpfr_set_si(epsilon2, 2, MPFR_RNDN);
    mpfr_nextabove(epsilon2);
    mpfr_sub_si(epsilon2, epsilon2, 2, MPFR_RNDN);
    mpfr_sqr(epsilon2, epsilon2, MPFR_RNDN);
300     // c = c_guess
    mpc_set(c, c_guess, MPC_RNDNN);
    for (int i = 0; i < maxsteps; ++i) {
      if (m_stepped != (result = m_r_nucleus_naive_step_raw(c, c, period, z, dc,↙
          ↳  c_new, d, d2, epsilon2))) {
```

```
        break;
      }
    }
    // c_out = c;
    mpc_set_prec(c_out, prec);
    mpc_set(c_out, c, MPC_RNDNN);
    // cleanup
    mpc_clear(c);
    mpc_clear(z);
    mpc_clear(dc);
    mpc_clear(c_new);
    mpc_clear(d);
    mpfr_clear(d2);
    mpfr_clear(epsilon2);
  }
  return result;
}

extern m_newton m_r_nucleus_step_raw(mpc_t c_out, const mpc_t c_guess, int ↙
    ↘ period, mpfr_t epsilon2, mpc_t c_new, mpc_t z, mpc_t dc, mpc_t h, mpc_t dh↙
    ↘ , mpc_t f, mpc_t df)
{
  mpc_set_dc(z, 0, MPC_RNDNN);
  mpc_set_dc(dc, 0, MPC_RNDNN);
  mpc_set_dc(h, 1, MPC_RNDNN);
  mpc_set_dc(dh, 0, MPC_RNDNN);
  for (int i = 1; i <= period; ++i) {
    // dc = 2 * z * dc + 1;
    mpc_mul(dc, z, dc, MPC_RNDNN);
    mpc_mul_2si(dc, dc, 1, MPC_RNDNN);
    mpfr_add_d(mpc_realref(dc), mpc_realref(dc), 1, MPFR_RNDN);
    // z = z * z + c_guess;
    mpc_sqr(z, z, MPC_RNDNN);
    mpc_add(z, z, c_guess, MPC_RNDNN);
    // reject lower periods
    if (i < period && period % i == 0)
    {
      // h = h * z;
      mpc_mul(h, h, z, MPC_RNDNN);
      // dh = dh + dc / z;
      mpc_div(f, dc, z, MPC_RNDNN);
      mpc_add(dh, dh, f, MPC_RNDNN);
    }
  }
  // build function
  // dh = dh * h;
  mpc_mul(dh, dh, h, MPC_RNDNN);
  // df = dc * h - z * dh
  mpc_mul(dc, dc, h, MPC_RNDNN);
  mpc_mul(dh, z, dh, MPC_RNDNN);
  mpc_sub(df, dc, dh, MPC_RNDNN);
  // f = z / h;
  mpc_div(f, z, h, MPC_RNDNN);
  // df /= h * h
  mpc_sqr(h, h, MPC_RNDNN);
  mpc_div(df, df, h, MPC_RNDNN);
  // newton step
```

```
  // c_new = c_guess - f / df;
  mpc_div(f, f, df, MPC_RNDNN);
  mpc_sub(c_new, c_guess, f, MPC_RNDNN);
  // check convergence
  // f = c_new - c_guess;
  mpc_sub(f, c_new, c_guess, MPC_RNDNN);
  mpc_norm(mpc_realref(df), f, MPC_RNDNN);
  if (mpfr_lessequal_p(mpc_realref(df), epsilon2)) {
    mpc_set(c_out, c_new, MPC_RNDNN);
    return m_converged;
  }
  if (mpfr_number_p(mpc_realref(f)) && mpfr_number_p(mpc_imagref(f))) {
    mpc_set(c_out, c_new, MPC_RNDNN);
    return m_stepped;
  } else {
    mpc_set(c_out, c_guess, MPC_RNDNN);
    return m_failed;
  }
}

extern m_newton m_r_nucleus(mpc_t c_out, const mpc_t c_guess, int period, int maxsteps, int ncpus) {
  m_newton result = m_failed;
  // prec
  mpfr_prec_t precr, preci, prec;
  mpc_get_prec2(&precr, &preci, c_guess);
  prec = precr > preci ? precr : preci;
  // init
  {
    (void) ncpus;
    mpc_t c, z, dc, c_new, h, dh, f, df;
    mpfr_t epsilon2;
    mpc_init2(c, prec);
    mpc_init2(z, prec);
    mpc_init2(dc, prec);
    mpc_init2(c_new, prec);
    mpc_init2(h, prec);
    mpc_init2(dh, prec);
    mpc_init2(f, prec);
    mpc_init2(df, prec);
    mpfr_init2(epsilon2, prec);
    // epsilon
    mpfr_set_si(epsilon2, 2, MPFR_RNDN);
    mpfr_nextabove(epsilon2);
    mpfr_sub_si(epsilon2, epsilon2, 2, MPFR_RNDN);
    mpfr_sqr(epsilon2, epsilon2, MPFR_RNDN);
    // c = c_guess
    mpc_set(c, c_guess, MPC_RNDNN);
    for (int i = 0; i < maxsteps; ++i)
      if (m_stepped != (result = m_r_nucleus_step_raw(c, c, period, epsilon2, c_new, z, dc, h, dh, f, df)))
        break;
    // c_out = c;
    mpc_set_prec(c_out, prec);
    mpc_set(c_out, c, MPC_RNDNN);
    // cleanup
    mpc_clear(c);
```

```
    mpc_clear(z);
    mpc_clear(dc);
    mpc_clear(c_new);
    mpc_clear(h);
    mpc_clear(dh);
    mpc_clear(f);
    mpc_clear(df);
    mpfr_clear(epsilon2);
  }
  return result;
}
```

73 c/lib/m_r_parent.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html

#include <mandelbrot-numerics.h>

extern int m_r_parent(mpq_t angle_out, mpc_t root_out, mpc_t parent_out, const mpc_t nucleus, int period, int maxsteps, int ncpus) {
  int retval = -1;
  // prec
  mpfr_prec_t precr, preci, prec;
  mpc_get_prec2(&precr, &preci, nucleus);
  prec = precr > preci ? precr : preci;
  switch (m_r_shape(nucleus, period)) {
    case m_cardioid: { // find root directly
      // init
      mpc_t z, c, i;
      mpc_init2(z, prec);
      mpc_init2(c, prec);
      mpc_init2(i, prec);
      // z = nucleus; c = nucleus; i = 1;
      mpc_set(z, nucleus, MPC_RNDNN);
      mpc_set(c, nucleus, MPC_RNDNN);
      mpc_set_si(i, 1, MPC_RNDNN);
      m_r_interior(z, c, z, c, i, period, maxsteps);
      // root_out = c;
      mpc_set_prec(root_out, prec);
      mpc_set(root_out, c, MPC_RNDNN);
      // cleanup
      mpc_clear(z);
      mpc_clear(c);
      mpc_clear(i);
      retval = 0; // no parent
      break;
    }
    case m_circle: { // trace internal ray to near to root
      // init
      mpc_t z, c, i, root0, root1, parent_guess, w0, w, dw, interior;
      mpfr_t mz2, z2, d, pi;
      mpc_init2(z, prec);
      mpc_init2(c, prec);
      mpc_init2(i, prec);
      mpc_init2(root0, prec);
```

```
    mpc_init2(root1, prec);
    mpc_init2(parent_guess, prec);
    mpc_init2(w0, prec);
    mpc_init2(w, prec);
    mpc_init2(dw, prec);
    mpc_init2(interior, prec);
    mpfr_init2(mz2, prec);
    mpfr_init2(z2, prec);
    mpfr_init2(d, prec);
    mpfr_init2(pi, prec);
    mpfr_const_pi(pi, MPFR_RNDN);
    // z = nucleus; c = nucleus;
    mpc_set(z, nucleus, MPC_RNDNN);
    mpc_set(c, nucleus, MPC_RNDNN);
    for (int step = 0; step < maxsteps - 1; ++step) {
      // i = (step + 0.5) / maxsteps;
      mpc_set_d(i, (step + 0.5) / maxsteps, MPC_RNDNN);
      m_r_interior(z, c, z, c, i, period, maxsteps);
    }
    // root0 = c;
    mpc_set(root0, c, MPC_RNDNN);
    // i = (maxsteps - 0.5) / maxsteps;
    mpc_set_d(i, (maxsteps - 0.5) / maxsteps, MPC_RNDNN);
    m_r_interior(z, c, z, c, i, period, maxsteps);
    // root1 = c;
    mpc_set(root1, c, MPC_RNDNN);
    // find interior coordinate of a point just past the root into the parent
    // parent_guess = 2 * root1 - root0;
    mpc_mul_2si(root1, root1, 1, MPC_RNDNN);
    mpc_sub(parent_guess, root1, root0, MPC_RNDNN);
    // c = parent_guess; z = 0; mz2 = 1.0 / 0.0;
    mpc_set(c, parent_guess, MPC_RNDNN);
    mpc_set_si(z, 0, MPC_RNDNN);
    mpfr_set_d(mz2, 1.0 / 0.0, MPFR_RNDN);
    for (int p = 1; p < period; ++p) {
      // z = z * z + c;
      mpc_sqr(z, z, MPC_RNDNN);
      mpc_add(z, z, c, MPC_RNDNN);
      // z2 = norm(z);
      mpc_norm(z2, z, MPFR_RNDN);
      if (mpfr_less_p(z2, mz2)) {
        // mz2 = z2;
        mpfr_set(mz2, z2, MPFR_RNDN);
        if (period % p == 0) {
          m_r_attractor(w0, z, c, p, maxsteps);
          // w = w0; dw = 1;
          mpc_set(w, w0, MPC_RNDNN);
          mpc_set_si(dw, 1, MPC_RNDNN);
          for (int q = 0 ; q < p; ++q) {
            // dw = 2 * w * dw;
            mpc_mul(dw, w, dw, MPC_RNDNN);
            mpc_mul_2si(dw, dw, 1, MPC_RNDNN);
            // w = w * w + c;
            mpc_sqr(w, w, MPC_RNDNN);
            mpc_add(w, w, c, MPC_RNDNN);
          }
          // d = norm(dw) - 1;
```

```
100                 mpc_norm(d, dw, MPFR_RNDN);
                    mpfr_sub_ui(d, d, 1, MPFR_RNDN);
                    if (mpfr_sgn(d) <= 0) {
                      // interior to component of period p
                      int den = period / p;
105                   // num = ((int) round(den * carg(dw) / twopi) + den) % den;
                      mpc_arg(d, dw, MPFR_RNDN);
                      mpfr_div(d, d, pi, MPFR_RNDN);
                      mpfr_div_2ui(d, d, 1, MPFR_RNDN);
                      mpfr_mul_si(d, d, den, MPFR_RNDN);
110                   mpfr_round(d, d);
                      int num = (mpfr_get_si(d, MPFR_RNDN) + den) % den;
                      mpq_set_si(angle_out, num, den);
                      mpq_canonicalize(angle_out);
                      m_r_nucleus(c, c, p, maxsteps, ncpus);
115                   // parent_out = c;
                      mpc_set(parent_out, c, MPC_RNDNN);
                      // interior = cexp(I * twopi * num / (double) den);
                      mpfr_mul_d(d, pi, 2 * mpq_get_d(angle_out), MPFR_RNDN);
                      mpfr_sin_cos(mpc_imagref(interior), mpc_realref(interior), d, ↙
                          ↳ MPFR_RNDN);
120                   m_r_interior(w, c, w0, c, interior, p, maxsteps);
                      // root_out = c;
                      mpc_set(root_out, c, MPC_RNDNN);
                      retval = p; // period of parent
                      break;
125                 }
                  }
                }
              }
              // cleanup
130           mpc_clear(z);
              mpc_clear(c);
              mpc_clear(i);
              mpc_clear(root0);
              mpc_clear(root1);
135           mpc_clear(parent_guess);
              mpc_clear(w0);
              mpc_clear(w);
              mpc_clear(dw);
              mpc_clear(interior);
140           mpfr_clear(mz2);
              mpfr_clear(z2);
              mpfr_clear(d);
              mpfr_clear(pi);
              break;
145         }
          }
          return retval; // fail
        }
```

74 c/lib/m_r_shape.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html
```

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

extern void m_r_shape_estimate(mpc_t shape, const mpc_t nucleus, int period) {
  // prec
  mpfr_prec_t precr, preci, prec;
  mpc_get_prec2(&precr, &preci, nucleus);
  prec = precr > preci ? precr : preci;
  // init
  mpc_t z, dc, dz, dcdc, dcdz, t;
  mpc_init2(z, prec);
  mpc_init2(dc, prec);
  mpc_init2(dz, prec);
  mpc_init2(dcdc, prec);
  mpc_init2(dcdz, prec);
  mpc_init2(t, prec);
  // z = nucleus; dc = 1; dz = 1; dcdc = 0; dcdz = 0;
  mpc_set(z, nucleus, MPC_RNDNN);
  mpc_set_si(dc, 1, MPC_RNDNN);
  mpc_set_si(dz, 1, MPC_RNDNN);
  mpc_set_si(dcdc, 0, MPC_RNDNN);
  mpc_set_si(dcdz, 0, MPC_RNDNN);
  for (int i = 1; i < period; ++i) {
    // dcdc = 2 * (z * dcdc + dc * dc);
    mpc_mul(dcdc, z, dcdc, MPC_RNDNN);
    mpc_sqr(t, dc, MPC_RNDNN);
    mpc_add(dcdc, dcdc, t, MPC_RNDNN);
    mpc_mul_2si(dcdc, dcdc, 1, MPC_RNDNN);
    // dcdz = 2 * (z * dcdz + dc * dz);
    mpc_mul(dcdz, z, dcdz, MPC_RNDNN);
    mpc_mul(t, dc, dz, MPC_RNDNN);
    mpc_add(dcdz, dcdz, t, MPC_RNDNN);
    mpc_mul_2si(dcdz, dcdz, 1, MPC_RNDNN);
    // dc = 2 * z * dc + 1;
    mpc_mul(dc, z, dc, MPC_RNDNN);
    mpc_mul_2si(dc, dc, 1, MPC_RNDNN);
    mpc_add_ui(dc, dc, 1, MPC_RNDNN);
    // dz = 2 * z * dz;
    mpc_mul(dz, z, dz, MPC_RNDNN);
    mpc_mul_2si(dz, dz, 1, MPC_RNDNN);
    // z = z * z + nucleus;
    mpc_sqr(z, z, MPC_RNDNN);
    mpc_add(z, z, nucleus, MPC_RNDNN);
  }
  // shape = -(dcdc / (2 * dc) + dcdz / dz) / (dc * dz);
  mpc_div(dcdc, dcdc, dc, MPC_RNDNN);
  mpc_div_2si(dcdc, dcdc, 1, MPC_RNDNN);
  mpc_div(dcdz, dcdz, dz, MPC_RNDNN);
  mpc_add(dcdc, dcdc, dcdz, MPC_RNDNN);
  mpc_mul(dc, dc, dz, MPC_RNDNN);
  mpc_div(shape, dcdc, dc, MPC_RNDNN);
  mpc_neg(shape, shape, MPC_RNDNN);
  // cleanup
  mpc_clear(z);
  mpc_clear(dc);
  mpc_clear(dz);
  mpc_clear(dcdc);
  mpc_clear(dcdz);
```

```
  mpc_clear(t);
}

extern m_shape m_r_shape_discriminant(const mpc_t shape) {
  // prec
  mpfr_prec_t precr, preci, prec;
  mpc_get_prec2(&precr, &preci, shape);
  prec = precr > preci ? precr : preci;
  // shape1 = shape - 1
  mpc_t one, shape1;
  mpc_init2(one, prec);
  mpc_init2(shape1, prec);
  mpc_set_si(one, 1, MPC_RNDNN);
  mpc_sub(shape1, shape, one, MPC_RNDNN);
  // e, e1
  mpfr_t e, e1;
  mpfr_init2(e, prec);
  mpfr_init2(e1, prec);
  // e = norm(shape); e1 = norm(shape1);
  mpc_norm(e, shape, MPFR_RNDN);
  mpc_norm(e1, shape1, MPFR_RNDN);
  m_shape retval;
  if (mpfr_less_p(e, e1)) {
    retval = m_cardioid;
  } else {
    retval = m_circle;
  }
  // cleanup
  mpc_clear(one);
  mpc_clear(shape1);
  mpfr_clear(e);
  mpfr_clear(e1);
  return retval;
}

extern m_shape m_r_shape(const mpc_t nucleus, int period) {
  // prec
  mpfr_prec_t precr, preci, prec;
  mpc_get_prec2(&precr, &preci, nucleus);
  prec = precr > preci ? precr : preci;
  // shape
  mpc_t shape;
  mpc_init2(shape, prec);
  // retval = discriminant(estimate(nucleus, period))
  m_r_shape_estimate(shape, nucleus, period);
  m_shape retval = m_r_shape_discriminant(shape);
  // cleanup
  mpc_clear(shape);
  return retval;
}
```

75 c/lib/m_r_size.c

```
// mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
// Copyright (C) 2015-2018 Claude Heiland-Allen
// License GPL3+ http://www.gnu.org/licenses/gpl.html
```

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

extern void m_r_size(mpc_t size, const mpc_t nucleus, int period) {
  // prec
  mpfr_prec_t precr, preci, prec;
  mpc_get_prec2(&precr, &preci, nucleus);
  prec = precr > preci ? precr : preci;
  // init
  mpc_t z, l, l1, b;
  mpc_init2(z, prec);
  mpc_init2(l, 53);
  mpc_init2(l1, 53);
  mpc_init2(b, 53);
  // l = 1; b = 1; z = 0;
  mpc_set_si(l, 1, MPC_RNDNN);
  mpc_set_si(b, 1, MPC_RNDNN);
  mpc_set_si(z, 0, MPC_RNDNN);
  for (int i = 1; i < period; ++i) {
    // z = z * z + nucleus;
    mpc_sqr(z, z, MPC_RNDNN);
    mpc_add(z, z, nucleus, MPC_RNDNN);
    // l = 2 * z * l;
    mpc_mul(l, z, l, MPC_RNDNN);
    mpc_mul_2si(l, l, 1, MPC_RNDNN);
    // b = b + 1 / l;
    mpc_ui_div(l1, 1, l, MPC_RNDNN);
    mpc_add(b, b, l1, MPC_RNDNN);
  }
  // size = 1 / (b * l * l);
  mpc_set_prec(size, 53);
  mpc_sqr(l, l, MPC_RNDNN);
  mpc_mul(l1, b, l, MPC_RNDNN);
  mpc_ui_div(size, 1, l1, MPC_RNDNN);
  // cleanup
  mpc_clear(z);
  mpc_clear(l);
  mpc_clear(l1);
  mpc_clear(b);
}
```

76 c/lib/pkgconfig/mandelbrot-numerics.pc.in

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

Name: mandelbrot-numerics
Description: Numerical algorithms related to the Mandelbrot set
Version: 0.1.0.0
URL: https://code.mathr.co.uk/mandelbrot-numerics
Libs: -L${libdir} -lmandelbrot-numerics
Libs.private: -lmpc -lmpfr -lgmp -lm
Cflags: -I${includedir}
```

77 COPYING.md

```
### 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
630 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>
635
        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.
640
        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.
645
        You should have received a copy of the GNU General Public License
        along with this program.  If not, see <https://www.gnu.org/licenses/>.

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

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

655     <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.

660 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".

665 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 <https://www.gnu.org/licenses/>.

670 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,
675 please read <https://www.gnu.org/licenses/why-not-lgpl.html>.
```

78 .gitignore

```
## mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
## Copyright (C) 2015-2017 Claude Heiland-Allen
## License GPL3+ http://www.gnu.org/licenses/gpl.html

5   *.a
    *.d
```

```
*.dat
*.log
*.o
*.hi
*.pc
*.png
*.gif
*.so
*.js
*.js.mem
*.wav
c/bin/m-box-period
c/bin/m-domain-size
c/bin/m-exray-in
c/bin/m-exray-out
c/bin/m-interior
c/bin/m-misiurewicz
c/bin/m-nucleus
c/bin/m-parent
c/bin/m-shape
c/bin/m-size
c/bin/m-attractor
c/bin/m-circles
c/bin/m-fibonacci-phi
c/bin/m-size-precision
c/bin/m-ball-period
c/bin/m-describe
c/bin/m-pi-rays
c/bin/m-feigenbaum
c/bin/m-two-spirals-out
c/bin/m-domain-coord
c/bin/m-ejs-tree-morph
c/bin/m-box-misiurewicz
c/bin/m-ejs-evotree-morph
c/bin/m-nearest-roots
c/bin/m-ball-vs-box-period
c/bin/m-furcation
c/bin/m-disc-period
c/bin/m-sonify
c/bin/m-feigenbaum3
c/bin/m-equipotential
c/bin/m-exray-out-perturbed
c/bin/m-feature-database
c/bin/m-golden-spiral
c/bin/m-rodney
c/bin/m-stability
hs/bin/m-stability
.cabal-sandbox
cabal.sandbox.config
dist
-
```

79 glsl/complex.glsl

```
//-------------------------------------------------------------
// complex number functions
```

```
vec2 cAdd( vec2 a, float s ) {
  return vec2( a.x+s, a.y );
}

vec2 cAdd( float s, vec2 a ) {
  return vec2( s+a.x, a.y );
}

vec2 cAdd( vec2 a, vec2 s ) {
  return a + s;
}

vec2 cSub( vec2 a, float s ) {
  return vec2( a.x-s, a.y );
}

vec2 cSub( float s, vec2 a ) {
  return vec2( s-a.x, -a.y );
}

vec2 cSub( vec2 a, vec2 s ) {
  return a - s;
}

vec2 cConj( vec2 z ) {
  return vec2(z.x,-z.y);
}

float cNorm(vec2 z) {
        return dot(z, z);
}

float cAbs(vec2 z) {
        return length(z);
}

float cArg(vec2 a) {
        return atan(a.y,a.x);
}

vec2 cSqr(vec2 z) {
        return vec2(z.x*z.x-z.y*z.y,2.*z.x*z.y);
}

vec2 cMul(vec2 a, vec2 b) {
        return vec2( a.x*b.x -  a.y*b.y,a.x*b.y + a.y * b.x);
}

vec2 cInverse(vec2 a) {
        return  vec2(a.x,-a.y)/dot(a,a);
}

vec2 cDiv( vec2 a, vec2 b ) {
  float d = dot(b,b);
  return vec2( dot(a,b), a.y*b.x - a.x*b.y ) / d;
}
```

```
vec2 cSqrt( vec2 z ) {
  float m = length(z);
  return sqrt( max(vec2(0.0), 0.5*vec2(m+z.x, m-z.x)) ) *
    vec2( 1.0, sign(z.y) );
}

vec2 cExp(vec2 z) {
        return exp(z.x) * vec2(cos(z.y), sin(z.y));
}

vec2 cLog(vec2 a) {
        return vec2(log(cAbs(a)),cArg(a));
}

vec2 cPow(vec2 z, vec2 a) {
        return cExp(cMul(cLog(z), a));
}

vec2 cPow(vec2 z, float a) {
        return cExp(cLog(z) * a);
}

vec2 cSin(vec2 z) {
        return vec2(sin(z.x)*cosh(z.y), cos(z.x)*sinh(z.y));
}

vec2 cCos(vec2 z) {
        return vec2(cos(z.x)*cosh(z.y), -sin(z.x)*sinh(z.y));
}

vec2 cTan(vec2 z) {
        return cDiv(cSin(z), cCos(z));
}

vec2 cSinh(vec2 z) {
  return 0.5 * (cExp(z) - cExp(-z));
}

vec2 cCosh(vec2 z) {
  return 0.5 * (cExp(z) + cExp(-z));
}

vec2 cTanh(vec2 z) {
        return cDiv(cSinh(z), cCosh(z));
}

vec2 cAsin(vec2 z) {
  const vec2 I = vec2(0.0, 1.0);
  return cMul(-I, cLog(cMul(I, z) + cSqrt(cSub(1.0, cSqr(z)))));
}

vec2 cAcos(vec2 z) {
  const vec2 I = vec2(0.0, 1.0);
  return cMul(-I, cLog(z + cMul(I, cSqrt(cSub(1.0, cSqr(z))))));
}

vec2 cAtan(vec2 z) {
```

```
  const vec2 I = vec2(0.0, 1.0);
  return cDiv
    ( cLog(cAdd(1.0, cMul(I, z))) - cLog(cSub(1.0, cMul(I, z)))
    , 2.0 * I
    );
}

vec2 cAsinh(vec2 z) {
  return cLog(z + cSqrt(cAdd(cSqr(z), 1.0)));
}

vec2 cAcosh(vec2 z) {
  return 2.0 *
    cLog(cSqrt(0.5 * cAdd(z, 1.0)) + cSqrt(0.5 * cSub(z, 1.0)));
}

vec2 cAtanh(vec2 z) {
  return 0.5 * (cLog(cAdd(1.0, z)) - cLog(cSub(1.0, z)));
}

//-----------------------------------------------------------------
// dual complex number functions for automatic differentiation

vec4 cConst( float z ) {
  return vec4( z, 0.0, 0.0, 0.0 );
}

vec4 cConst( vec2 z ) {
  return vec4( z, 0.0, 0.0 );
}

vec4 cVar( float z ) {
  return vec4( z, 0.0, 1.0, 0.0 );
}

vec4 cVar( vec2 z ) {
  return vec4( z, 1.0, 0.0 );
}

vec2 cVar( vec4 z ) {
  return z.xy;
}

vec2 cDeriv( vec4 z ) {
  return z.zw;
}

float cNorm(vec4 z) {
        return cNorm(z.xy);
}

float cAbs(vec4 z) {
        return cAbs(z.xy);
}

float cArg(vec4 z) {
        return cArg(z.xy);
```

```
}

vec4 cAdd( vec4 z, float a ) {
  return vec4( z.x + a, z.yzw );
}

vec4 cAdd( float a, vec4 z ) {
  return vec4( a + z.x, z.yzw );
}

vec4 cAdd( vec4 z, vec2 a ) {
  return vec4( z.xy + a, z.zw );
}

vec4 cAdd( vec2 a, vec4 z ) {
  return vec4( a + z.xy, z.zw );
}

vec4 cAdd( vec4 a, vec4 z ) {
  return a + z;
}

vec4 cSub( vec4 z, float a ) {
  return vec4( z.x - a, z.yzw );
}

vec4 cSub( float a, vec4 z ) {
  return vec4( a - z.x, -z.yzw );
}

vec4 cSub( vec4 z, vec2 a ) {
  return vec4( z.xy - a, z.zw );
}

vec4 cSub( vec2 a, vec4 z ) {
  return vec4( a - z.xy, -z.zw );
}

vec4 cSub( vec4 a, vec4 z ) {
  return a - z;
}

vec4 cSqr( vec4 z ) {
  return vec4( cSqr(z.xy), 2.0 * cMul(z.xy, z.zw) );
}

vec4 cMul( vec4 a, vec4 b ) {
  return vec4
    ( cMul(a.xy, b.xy)
    , cMul(a.xy, b.zw) + cMul(a.zw, b.xy)
    );
}

vec4 cMul( vec2 a, vec4 b ) {
  return cMul( cConst(a), b );
}
```

```
vec4 cMul( vec4 a, vec2 b ) {
  return cMul( a, cConst(b) );
}

vec4 cDiv( vec4 a, vec4 b ) {
  return vec4
    ( cDiv(a.xy, b.xy)
    , cDiv(cMul(a.zw, b.xy) - cMul(a.xy, b.zw), cSqr(b.xy))
    );
}

vec4 cDiv( float a, vec4 b ) {
  return cDiv( cConst(a), b );
}

vec4 cDiv( vec2 a, vec4 b ) {
  return cDiv( cConst(a), b );
}

vec4 cDiv( vec4 a, vec2 b ) {
  return cDiv( a, cConst(b) );
}

vec4 cInverse( vec4 a ) {
  return cDiv(1.0, a);
}

vec4 cSqrt( vec4 a ) {
  vec2 s = cSqrt(a.xy);
  return vec4( s, cDiv(a.zw, 2.0 * s) );
}

vec4 cExp( vec4 a ) {
  vec2 s = cExp(a.xy);
  return vec4( s, cMul(s, a.zw) );
}

vec4 cLog( vec4 a ) {
  return vec4( cLog(a.xy), cDiv(a.zw, a.xy) );
}

vec4 cSin( vec4 z ) {
  const vec2 I = vec2(0.0, 1.0);
  return cDiv(cExp(cMul(I, z)) - cExp(cMul(-I, z)), 2.0 * I);
}

vec4 cCos( vec4 z ) {
  const vec2 I = vec2(0.0, 1.0);
  return cExp(cMul(I, z)) + cExp(cMul(-I, z)) / 2.0;
}

vec4 cTan( vec4 a ) {
  return cDiv(cSin(a), cCos(a));
}

vec4 cSinh(vec4 z) {
  return 0.5 * (cExp(z) - cExp(-z));
```

```
}

vec4 cCosh(vec4 z) {
  return 0.5 * (cExp(z) + cExp(-z));
}

vec4 cTanh(vec4 z) {
        return cDiv(cSinh(z), cCosh(z));
}

vec4 cAsin(vec4 z) {
  const vec2 I = vec2(0.0, 1.0);
  return cMul(-I, cLog(cMul(I, z) + cSqrt(cSub(1.0, cSqr(z)))));
}

vec4 cAcos(vec4 z) {
  const vec2 I = vec2(0.0, 1.0);
  return cMul(-I, cLog(z + cMul(I, cSqrt(cSub(1.0, cSqr(z))))));
}

vec4 cAtan(vec4 z) {
  const vec2 I = vec2(0.0, 1.0);
  return cDiv
    ( cLog(cAdd(1.0, cMul(I, z))) - cLog(cSub(1.0, cMul(I, z)))
    , 2.0 * I
    );
}

vec4 cAsinh(vec4 z) {
  return cLog(z + cSqrt(cAdd(cSqr(z), 1.0)));
}

vec4 cAcosh(vec4 z) {
  return 2.0 *
    cLog(cSqrt(0.5 * cAdd(z, 1.0)) + cSqrt(0.5 * cSub(z, 1.0)));
}

vec4 cAtanh(vec4 z) {
  return 0.5 * (cLog(cAdd(1.0, z)) - cLog(cSub(1.0, z)));
}
```

80 hs/bin/m-stability.hs

```
{-# LANGUAGE FlexibleContexts #-}

import Data.Complex.Generic
import System.Environment (getArgs)
import System.Random

dot (x:+y) (u:+v) = x * u - y * v
mag2 z = dot z (conjugate z)

chunk :: Int -> [a] -> [[a]]
chunk n [] = []
chunk n zs = let (xs, ys) = splitAt n zs in xs : chunk n ys

exact = fmap toRational
```

```
inexact = fmap fromRational

corr :: [Complex Double] -> [Complex Double] -> Double
corr x y = ((/ fromIntegral (length x)) . sum $ zipWith dot x y) / (sqrt . (/ ↙
    ↪ fromIntegral (length x)) . sum . map mag2 $ zipWith (*) x y)

test :: Int -> Int -> Complex Double -> (Complex Double, Complex Double)
test m n c =
  let c' = exact c
      z0':z1s' = drop m $ iterate (\z -> z^2 + c') 0
      z1':_    = drop n z1s'
      z0 :z1s  = drop m $ iterate (\z -> z^2 + c ) 0
      z1:_     = drop n z1s
      e0 = inexact (z0' - exact z0)
      e1 = inexact (z1' - exact z1)
  in  (e0, e1)

main :: IO ()
main = do
  [m, n] <- map read <$> getArgs
  g <- newStdGen
  let r = uncurry corr . unzip . take 100 . map (test m n) . filter (\c -> dot c↙
      ↪  c <= 4) . map (\[x, y] -> x :+ y) . chunk 2 . randomRs (-2, 2) $ g
  print r
```

81 hs/lib/Mandelbrot/Numerics/BoxPeriod.hs

```
{-# LANGUAGE BangPatterns #-}
{-# LANGUAGE FlexibleContexts #-}
module Mandelbrot.Numerics.BoxPeriod
  ( boxPeriod
  ) where

import Mandelbrot.Numerics.Complex
import Mandelbrot.Numerics.Progress

cross
  :: Num r
  => Complex r -> Complex r -> r
cross (ax :+ ay) (bx :+ by) = ay * bx - ax * by

crossesPositiveRealAxis
  :: RealFloat r
  => Complex r -> Complex r -> Bool
crossesPositiveRealAxis a@(_:+ay) b@(_:+by)
  | signum ay /= signum by = s == t
  | otherwise = False
  where
    d@(_:+dy) = b - a
    s = signum dy
    t = signum (d `cross` a)

surroundsOrigin
  :: RealFloat r
  => Complex r -> Complex r -> Complex r -> Complex r -> Bool
surroundsOrigin a b c d
  = odd . length . filter id
```

```
  $ zipWith crossesPositiveRealAxis [a,b,c,d] [b,c,d,a]

boxPeriod
  :: (RealFloat r, Square r, Square (Complex r))
  => Complex r -> r -> Progress Int
boxPeriod c r = go 1 c0 c1 c2 c3
  where
    r' = negate r
    c0 = c + (r' :+ r')
    c1 = c + (r  :+ r')
    c2 = c + (r  :+ r )
    c3 = c + (r' :+ r )
    go !p z0 z1 z2 z3
      | notFinite' z0 = Failed p
      | notFinite' z1 = Failed p
      | notFinite' z2 = Failed p
      | notFinite' z3 = Failed p
      | surroundsOrigin z0 z1 z2 z3 = Done p
      | otherwise = Continue p $
          go (p + 1) (sqr z0 + c0) (sqr z1 + c1) (sqr z2 + c2) (sqr z3 + c3)
    notFinite' (x :+ y) = not (isNaN x) && not (isNaN y) && not (isInfinite x) ↙
      ↳ && not (isInfinite y)
```

82 hs/lib/Mandelbrot/Numerics/Child.hs

```
{-# LANGUAGE BangPatterns #-}
{-# LANGUAGE FlexibleContexts #-}
module Mandelbrot.Numerics.Child
  ( Child(..)
  , child
  , children
  , childSize
  ) where

import Data.Ratio
  ( (%)
  , denominator
  )
import Data.Strict.Tuple
  ( Pair(..)
  )

import Mandelbrot.Numerics.Complex
import Mandelbrot.Numerics.Interior
import Mandelbrot.Numerics.Nucleus
import Mandelbrot.Numerics.Shape
import Mandelbrot.Numerics.Size

data Child r = Child !Int !(Complex r)
  deriving (Eq, Read, Show)

childSize
  :: (RealFloat r, Square r, Square (Complex r), Approx r, Approx (Complex r))
  => Int -> Complex r -> Rational -> Maybe r
{-# SPECIALIZE childSize :: Int -> Complex Double -> Rational -> Maybe Double ↙
    ↳ #-}
childSize !p !c0 !t = case shape p c0 of
```

```
  Just Cardioid | finite s -> Just $! s * sin (pi * fromRational t) * 2
  Just Circle | finite s -> Just $! s
  _ -> Nothing
  where
    !s = magnitude (size p c0) / (q * q)
    !q = fromInteger (denominator t)

child
  :: (RealFloat r, Square r, Square (Complex r), Approx r, Approx (Complex r))
  => Int -> Complex r -> Rational -> Maybe (Child r)
{-# SPECIALIZE child :: Int -> Complex Double -> Rational -> Maybe (Child Double) #-}
child p c0 t = case childSize p c0 t of
  Just s -> case converge 2 64 $ interior p (l * b) c0 c0 of
    Just (_:!:root1) -> case converge 2 64 $ interior p b c0 c0 of
      Just (_:!:root0) ->
         let n = root0 + (0.5 * s :+ 0) * signum (root0 - root1)
         in  case converge 2 64 $ nucleus p1 n of
               Just c1 -> Just (Child p1 c1)
               _ -> Nothing
      _ -> Nothing
    _ -> Nothing
  _ -> Nothing
  where
    l = fromRational $ (denominator t - 1) % (denominator t)
    b = cis (2 * pi * fromRational t)
    p1 = p * fromInteger (denominator t)

children
  :: (RealFloat r, Square r, Square (Complex r), Approx r, Approx (Complex r))
  => Int -> Complex r -> [Rational] -> [Child r]
{-# SPECIALIZE children :: Int -> Complex Double -> [Rational] -> [Child Double] #-}
children _ _ [] = []
children p0 c0 (t:ts) = case child p0 c0 t of
  Just c@(Child p1 c1) -> c : children p1 c1 ts
  _ -> []
```

83 hs/lib/Mandelbrot/Numerics/Complex.hs

```
{- |
Complex number operations useful in Mandelbrot set numeric algorithms.
-}
module Mandelbrot.Numerics.Complex
  ( module Data.Complex
  , Approx(..)
  , converge
  , Square(..)
  , magnitudeSquared
  , loss
  ) where

import Data.Complex
import Data.Strict.Tuple
  ( Pair(..)
  )
import Safe
```

```
  ( lastMay
  )

class Approx t where
  finite :: t -> Bool
  finite = not . notFinite
  notFinite :: t -> Bool
  notFinite = not . finite
  bashZero :: t -> t
  approxEq :: Int -> t -> t -> Bool

instance Approx Double where
  notFinite x = isNaN x || isInfinite x
  bashZero x
    | 1 + abs x == 1 = 0
    | otherwise = x
  approxEq b u v
    | x == 0 && y == 0 = True
    | x >  0 && y >  0 = loss    x     y  > k
    | x <  0 && y <  0 = loss (-x) (-y) > k
    | otherwise = False
    where
      x = bashZero u
      y = bashZero v
      k = fromIntegral (floatDigits x - b)

loss :: Double -> Double -> Double
loss p q = negate . logBase 2 $ 1 - min p q / max p q

instance Approx t => Approx (Complex t) where
  {-# SPECIALIZE instance Approx (Complex Double) #-}
  finite (x :+ y) = finite x && finite y
  bashZero (x :+ y) = bashZero x :+ bashZero y
  approxEq b (ux :+ uy) (vx :+ vy) = approxEq b ux vx && approxEq b uy vy

instance (Approx s, Approx t) => Approx (Pair s t) where
  {-# SPECIALIZE instance Approx (Pair (Complex Double) (Complex Double)) #-}
  finite (a :!: b) = finite a && finite b
  bashZero (a :!: b) = bashZero a :!: bashZero b
  approxEq b (u :!: v) (x :!: y) = approxEq b u x && approxEq b v y

converge
  :: Approx t
  => Int -> Int -> [t] -> Maybe t
{-# SPECIALIZE converge :: Int -> Int -> [Complex Double] -> Maybe (Complex ↙
    ↳ Double) #-}
{-# SPECIALIZE converge :: Int -> Int -> [Pair (Complex Double) (Complex Double)↙
    ↳ ] -> Maybe (Pair (Complex Double) (Complex Double)) #-}
converge b n = lastMay . trim . filter finite . take n
  where
    trim [] = []
    trim [x] = [x]
    trim (x:ys@(y:_))
      | approxEq b x y = [x,y]
      | otherwise = trim ys

class Num t => Square t where
```

```
  sqr :: t -> t
  sqr x = x * x
  double :: t -> t
  double x = x + x

instance Square Double

instance (RealFloat t, Square t) => Square (Complex t) where
  sqr (x :+ y) = (sqr x - sqr y) :+ double (x * y)
  {-# SPECIALIZE instance Square (Complex Double) #-}
  double (x :+ y) = double x :+ double y

magnitudeSquared :: Square r => Complex r -> r
{-# SPECIALIZE magnitudeSquared :: Complex Double -> Double #-}
magnitudeSquared (x :+ y) = sqr x + sqr y
```

84 hs/lib/Mandelbrot/Numerics/DomainCoord.hs

```
{-# LANGUAGE BangPatterns #-}
{-# LANGUAGE FlexibleContexts #-}
module Mandelbrot.Numerics.DomainCoord
  ( domainCoord
  , atDomainCoord
  ) where

import Mandelbrot.Numerics.Complex

domainCoord
  :: (RealFloat r, Square r)
  => Int -> Int -> Complex r -> Complex r
{-# SPECIALIZE domainCoord :: Int -> Int -> Complex Double -> Complex Double #-}
domainCoord q p c = go1 0 0
  where
    go1 n !z
      | n == q = go2 n z
      | otherwise = go1 (n + 1) (sqr z + c)
      where
        go2 n !w
          | n == p = w / z
          | otherwise = go2 (n + 1) (sqr w + c)


atDomainCoord
  :: (RealFloat r, Square r)
  => Int -> Int -> Complex r -> Complex r -> [Complex r]
{-# SPECIALIZE atDomainCoord :: Int -> Int -> Complex Double -> Complex Double ↙
    ↳ -> [Complex Double] #-}
atDomainCoord q p g c = c : go1 0 0 0
  where
    go1 n !z !dz
      | n == q = go2 n z dz
      | otherwise = go1 (n + 1) (sqr z + c) (double (z * dz) + 1)
      where
        go2 n !w !dw
          | n == p = atDomainCoord q p g c'
          | otherwise = go2 (n + 1) (sqr w + c) (double (w * dw) + 1)
          where
```

```
            c' = c - f / df
            f = w / z - g
            df = (dw * z - w * dz) / sqr z
```

85 hs/lib/Mandelbrot/Numerics/DomainSize.hs

```
{-# LANGUAGE BangPatterns #-}
{-# LANGUAGE FlexibleContexts #-}
module Mandelbrot.Numerics.DomainSize
  ( domainSize
  ) where

import Mandelbrot.Numerics.Complex

domainSize
  :: (RealFloat r, Square r, Square (Complex r))
  => Int -> Complex r -> r
{-# SPECIALIZE domainSize :: Int -> Complex Double -> Double #-}
domainSize p c = go 1 (magnitudeSquared c) c 1
  where
    go q !zq2 !z !dc
      | q >= p = sqrt zq2 / magnitude dc
      | otherwise = go (q + 1) zq2' (sqr z + c) (double (z * dc) + 1)
      where
        zq2' = zq2 `min` magnitudeSquared z
```

86 hs/lib/Mandelbrot/Numerics.hs

```
module Mandelbrot.Numerics
  ( module Mandelbrot.Numerics.BoxPeriod
  , module Mandelbrot.Numerics.Child
  , module Mandelbrot.Numerics.Complex
  , module Mandelbrot.Numerics.DomainCoord
  , module Mandelbrot.Numerics.DomainSize
--  , module Mandelbrot.Numerics.ExRayIn
--  , module Mandelbrot.Numerics.ExRayOut
  , module Mandelbrot.Numerics.Interior
  , module Mandelbrot.Numerics.Misiurewicz
  , module Mandelbrot.Numerics.Nucleus
  , module Mandelbrot.Numerics.Parent
  , module Mandelbrot.Numerics.Progress
  , module Mandelbrot.Numerics.Shape
  , module Mandelbrot.Numerics.Size
  , module Mandelbrot.Numerics.Wucleus
  ) where

import Mandelbrot.Numerics.BoxPeriod
import Mandelbrot.Numerics.Child
import Mandelbrot.Numerics.Complex
import Mandelbrot.Numerics.DomainCoord
import Mandelbrot.Numerics.DomainSize
--import Mandelbrot.Numerics.ExRayIn
--import Mandelbrot.Numerics.ExRayOut
import Mandelbrot.Numerics.Interior
import Mandelbrot.Numerics.Misiurewicz
import Mandelbrot.Numerics.Nucleus
```

```
import Mandelbrot.Numerics.Parent
import Mandelbrot.Numerics.Progress
import Mandelbrot.Numerics.Shape
import Mandelbrot.Numerics.Size
import Mandelbrot.Numerics.Wucleus
```

87 hs/lib/Mandelbrot/Numerics/Interior.hs

```
{-# LANGUAGE BangPatterns #-}
{-# LANGUAGE FlexibleContexts #-}
module Mandelbrot.Numerics.Interior
  ( interior
  ) where

import Data.Strict.Tuple
  ( Pair(..)
  )

import Mandelbrot.Numerics.Complex

interior
  :: (RealFloat r, Square r, Square (Complex r), Approx r, Approx (Complex r))
  => Int -> Complex r -> Complex r -> Complex r -> [Pair (Complex r) (Complex r)]
{-# SPECIALIZE interior :: Int -> Complex Double -> Complex Double -> Complex Double -> [Pair (Complex Double) (Complex Double)] #-}
interior !p !i !z0 !c0
  | p > 0 && finite i && finite z0 && finite c0 = go 0 z0 1 0 0 0
  | otherwise = []
  where
    go !q !z !dz !dc !dzdz !dcdz
      | q >= p = (z' :!: c') : interior p i z' c'
      | otherwise =
          go (q + 1) (sqr z + c0) (double (z * dz)) (double (z * dc) + 1)
            (double (z * dzdz + sqr dz)) (double (z * dcdz + dc * dz))
      where
        det = (dz - 1) * dcdz - dc * dzdz;
        z' = z0 - (dcdz * (z - z0) - dc * (dz - i)) / det;
        c' = c0 - ((dz - 1) * (dz - i) - dzdz * (z - z0)) / det;
```

88 hs/lib/Mandelbrot/Numerics/Misiurewicz.hs

```
{-# LANGUAGE BangPatterns #-}
{-# LANGUAGE FlexibleContexts #-}
module Mandelbrot.Numerics.Misiurewicz
  ( misiurewicz
  , misiurewiczNaive
  ) where

import Data.List
  ( foldl'
  )

import Data.Strict.Tuple
  ( Pair((:!:))
  )
```

```
15
    import Mandelbrot.Numerics.Complex

    misiurewiczNaive
      :: (RealFloat r, Square r, Square (Complex r), Approx r, Approx (Complex r))
20    => Int -> Int -> Complex r -> [Complex r]
    {-# SPECIALIZE misiurewiczNaive :: Int -> Int -> Complex Double -> [Complex ↙
        ↳ Double] #-}
    misiurewiczNaive pp p c0
      | pp > 0 && p > 0 && finite c0 = go 0 0 0 0 0 0
      | otherwise = []
25    where
        go i !zp !dcp !z !dc
          | i == pp + p = c' : misiurewiczNaive pp p c'
          | i == pp = go (i + 1) z dc (sqr z + c0) (double (z * dc) + 1)
          | otherwise = go (i + 1) zp dcp (sqr z + c0) (double (z * dc) + 1)
30        where
            c' = c0 - (z - zp) / (dc - dcp)

    misiurewicz
      :: (RealFloat r, Square r, Square (Complex r), Approx r, Approx (Complex r))
35    => Int -> Int -> Complex r -> [Complex r]
    {-# SPECIALIZE misiurewicz :: Int -> Int -> Complex Double -> [Complex Double] ↙
        ↳ #-}
    misiurewicz pp p c0
      | pp > 0 && p > 0 && finite c0 = go 0 [] 0 0
      | otherwise = []
40    where
        go i !ps !z !dc
          | i == pp + p = c' : misiurewicz pp p c'
          | otherwise = go (i + 1) ((z:!:dc):ps) (sqr z + c0) (double (z * dc + 1))
          where
45          c' = c0 - f / df
            f = g / h
            df = (dg * h - g * dh) / (h * h)
            g = z - zp
            dg = dc - dcp
50          zp :!: dcp = ps !! (p - 1)
            h :!: dh0 = foldl' prodsum (1 :!: 0) $ zipWith sub ps (drop p ps)
            dh = h * dh0
            sub (xp :!: xs) (yp :!: ys) = dp :!: ((xs - ys) / dp)
              where dp = xp - yp
55          prodsum (xp :!: xs) (yp :!: ys) = (xp * yp) :!: (xs + ys)
```

89 hs/lib/Mandelbrot/Numerics/Nucleus.hs

```
    {-# LANGUAGE BangPatterns #-}
    {-# LANGUAGE FlexibleContexts #-}
    module Mandelbrot.Numerics.Nucleus
      ( nucleus
5     ) where

    import Mandelbrot.Numerics.Complex

    nucleus
10    :: (RealFloat r, Square r, Square (Complex r))
      => Int -> Complex r -> [Complex r]
```

```
{-# SPECIALIZE nucleus :: Int -> Complex Double -> [Complex Double] #-}
nucleus !p !c0@(cx :+ cy)
  | p > 0 && fin cx && fin cy = go 0 0 0
  | otherwise = []
  where
    fin x = not (isNaN x) && not (isInfinite x)
    go !i !z !dc
      | i == p = c' : nucleus p c'
      | otherwise = go (i + 1) (sqr z + c0) (double (z * dc) + 1)
      where
        c' = c0 - z / dc
```

90 hs/lib/Mandelbrot/Numerics/Parent.hs

```
{-# LANGUAGE BangPatterns #-}
{-# LANGUAGE FlexibleContexts #-}
module Mandelbrot.Numerics.Parent
  ( Parent(..)
  , parent
  , parents
  ) where

import Data.Ratio
  ( (%)
  )
import Data.Strict.Tuple
  ( Pair(..)
  )

import Mandelbrot.Numerics.Complex
import Mandelbrot.Numerics.Interior
import Mandelbrot.Numerics.Nucleus
import Mandelbrot.Numerics.Shape
import Mandelbrot.Numerics.Wucleus

data Parent r
  = NoParent !(Complex r)
  | Parent !(Complex r) !Rational !Int !(Complex r)
  deriving (Eq, Read, Show)

parent
  :: (RealFloat r, Square r, Square (Complex r), Approx r, Approx (Complex r))
  => Int -> Complex r -> Maybe (Parent r)
{-# SPECIALIZE parent :: Int -> Complex Double -> Maybe (Parent Double) #-}
parent p c0 = case shape p c0 of
  Nothing -> Nothing
  Just Cardioid -> case converge 2 64 $ interior p 1 c0 c0 of
    Nothing -> Nothing
    Just (_:!:r) -> Just (NoParent r)
  Just Circle -> case converge 2 64 $ interior p ((k-3)/k) c0 c0 of
    Just (z1:!:c1) -> case converge 2 64 $ interior p ((k-1)/k) z1 c1 of
      Just (_:!:c2) -> go (double c2 - c1) 1 (1/0) 0
      _ -> Nothing
    _ -> Nothing
    where
      k = 32
      go c1 q zq z
```

```
        | q >= p = Nothing
        | zq' < zq && p `mod` q == 0 = case converge 2 64 $ wucleus q c1 z' of
            Just z1 -> case zdz q c1 z1 1 of
              (_:!:dz1) | magnitudeSquared dz1 <= 1 ->
                let den = p `div` q
                    num = round (fromIntegral den * phase dz1 / (2 * pi)) `mod` ↙
                        ↳ den
                    t = toInteger num % toInteger den
                    i = cis (2 * pi * fromRational t)
                in  case converge 2 64 $ nucleus q c1 of
                      Just c2 -> case converge 2 64 $ interior q i c2 c2 of
                        Just (_:!:c3) -> Just $ Parent c3 t q c2
                        _ -> Nothing
                      _ -> Nothing
              _ -> go c1 (q + 1) zq' z'
            _ -> go c1 (q + 1) zq' z'
        | zq' < zq = go c1 (q + 1) zq' z'
        | otherwise = go c1 (q + 1) zq z'
        where
          z' = sqr z + c1
          zq' = magnitudeSquared z'
    where
      zdz q c !z !dz
        | q == 0 = z :!: dz
        | otherwise = zdz (q - 1) c (sqr z + c) (double (z * dz))

parents
  :: (RealFloat r, Square r, Square (Complex r), Approx r, Approx (Complex r))
  => Int -> Complex r -> [Parent r]
{-# SPECIALIZE parents :: Int -> Complex Double -> [Parent Double] #-}
parents p c = case parent p c of
  Nothing -> []
  Just q@(NoParent _) -> [q]
  Just q@(Parent _ _ p' c') -> q : parents p' c'
```

91 hs/lib/Mandelbrot/Numerics/Progress.hs

```
module Mandelbrot.Numerics.Progress
  ( Progress(..)
  , skip
  ) where

data Progress a = Failed !a | Continue !a (Progress a) | Done !a
  deriving (Eq, Ord, Read, Show)

skip :: Int -> Progress a -> Progress a
skip n p
  | n <= 0 = p
  | otherwise = case p of
      Continue _ p' -> skip (n - 1) p'
      _ -> p
```

92 hs/lib/Mandelbrot/Numerics/Shape.hs

```
{-# LANGUAGE BangPatterns #-}
{-# LANGUAGE FlexibleContexts #-}
```

```
module Mandelbrot.Numerics.Shape
  ( Shape(..)
  , shape
  ) where

import Mandelbrot.Numerics.Complex

data Shape = Cardioid | Circle
  deriving (Eq, Ord, Enum, Bounded, Read, Show)

shape
  :: (RealFloat r, Square r, Square (Complex r), Approx r, Approx (Complex r))
  => Int -> Complex r -> Maybe Shape
{-# SPECIALIZE shape :: Int -> Complex Double -> Maybe Shape #-}
shape p c
  | p < 1 || notFinite e = Nothing
  | magnitudeSquared e < magnitudeSquared (e - 1) = Just Cardioid
  | otherwise = Just Circle
  where
    e = go 1 c 1 1 0 0
    go i !z !dc !dz !dcdc !dcdz
      | i == p = - (dcdc / (double dc) + dcdz / dz) / (dc * dz)
      | otherwise = go
          (i + 1) (sqr z + c) (double (z * dc) + 1) (double (z * dz))
          (double (z * dcdc + sqr dc)) (double (z * dcdz + dc * dz))
```

93 hs/lib/Mandelbrot/Numerics/Size.hs

```
{-# LANGUAGE BangPatterns #-}
{-# LANGUAGE FlexibleContexts #-}
module Mandelbrot.Numerics.Size
  ( size
  ) where

import Mandelbrot.Numerics.Complex

size
  :: (RealFloat r, Square r, Square (Complex r))
  => Int -> Complex r -> Complex r
{-# SPECIALIZE size :: Int -> Complex Double -> Complex Double #-}
size !p !c
  | p == 1 = 1
  | otherwise = go 2 c 0 (double c)
  where
    go !q !z !b !l
      | q == p = recip ((b + recip l) * sqr l)
      | otherwise = go (q + 1) (sqr z + c) (b + recip l) (double (z * l))
```

94 hs/lib/Mandelbrot/Numerics/Wucleus.hs

```
{-# LANGUAGE BangPatterns #-}
{-# LANGUAGE FlexibleContexts #-}
module Mandelbrot.Numerics.Wucleus
  ( wucleus
  ) where
```

```
import Mandelbrot.Numerics.Complex

wucleus
  :: (RealFloat r, Square r, Square (Complex r), Approx r, Approx (Complex r))
  => Int -> Complex r -> Complex r -> [Complex r]
{-# SPECIALIZE wucleus :: Int -> Complex Double -> Complex Double -> [Complex ↙
    ↳ Double] #-}
wucleus p c0 z0
  | p < 1 || notFinite c0 || notFinite z0 = []
  | otherwise = go 0 z0 1
  where
    go i !z !dz
      | i == p = z' : wucleus p c0 z'
      | otherwise = go (i + 1) (sqr z + c0) (double (z * dz))
      where
        z' = z0 - (z - z0) / (dz - 1)
```

95 hs/test/mandelbrot-numerics-tests.hs

```
{-# LANGUAGE TemplateHaskell #-}
module Main ( main ) where

import Data.Ratio
  ( (%)
  , denominator
  )
import Data.Word
  ( Word8
  , Word16
  )
import System.Exit
  ( exitFailure
  , exitSuccess
  )
import Test.QuickCheck
  ( Arbitrary(..)
  , arbitrarySizedBoundedIntegral
  , choose
  , vector
  , elements
  , Property
  , property
  , (==>)
  , quickCheckAll
  )

import Mandelbrot.Numerics

data Island = Island !Int !(Complex Double)
  deriving Show

islands :: [Island]
islands =
  [ Island 1 0
  , Island 3 (-1.754877666246693)
  , Island 4 ((-0.15652016683375508) :+ 1.0322471089228318)
  , Island 4 (-1.9407998065294847)
```

```
    , Island 4 ((-0.15652016683375508) :+ (-1.0322471089228318))
    ]

newtype IA8 = IA8{ unIA8 :: Rational } deriving Show
newtype IA16 = IA16 Rational deriving Show
newtype IA8s = IA8s [IA8] deriving Show

instance Arbitrary Island where
  arbitrary = elements islands

instance Arbitrary IA8 where
  arbitrary = do
    a <- arbitrarySizedBoundedIntegral
    b <- arbitrarySizedBoundedIntegral
    let n = min a b
        d = max a b
        fi :: Word8 -> Integer
        fi = fromIntegral
    return . IA8 $ (fi n + 1) % (fi d + 1)

instance Arbitrary IA16 where
  arbitrary = do
    a <- arbitrarySizedBoundedIntegral
    b <- arbitrarySizedBoundedIntegral
    let n = min a b
        d = max a b
        fi :: Word16 -> Integer
        fi = fromIntegral
    return . IA16 $ (fi n + 1) % (fi d + 1)

instance Arbitrary IA8s where
  arbitrary = do
    len <- choose (1, 10)
    IA8s `fmap` vector len

prop_parent_child :: Property
prop_parent_child = property $ \(IA16 s) (Island np n) ->
  0 < s && s < 1 ==>
  case child np n s of
    Just (Child p c) -> case parent p c of
      Just (Parent _ t _ _) -> s == t
      _ -> False
    _ -> False

prop_parents_children :: Property
prop_parents_children = property $ \(IA8s xs) (Island np n) ->
  let ss = map unIA8 xs
      cs = children np n ss
      Child p c = last cs
      ps = parents p c
      ts = reverse [ t | Parent _ t _ _ <- ps ]
  in  all (\s -> 0 < s && s < 1) ss &&
      product (map denominator ss) * fromIntegral np <= 2^(20 :: Int) ==>
      ss == ts

return []
main :: IO ()
```

```
main = do
  r <- $quickCheckAll
  if r then exitSuccess else exitFailure
```

96 INSTALL-emscripten.md

```
# Installing with Emscripten

See <https://stackoverflow.com/a/43583154/7459445> for instructions on setting
up a 32bit chroot for compiling emscripten, gmp, mpfr, mpc.  Then do this for
mandelbrot-numerics:

    git clone https://code.mathr.co.uk/mandelbrot-numerics.git
    cd mandelbrot-numerics
    make -C c/lib CC="emcc -I${HOME}/opt/include" prefix=${HOME}/opt -j 6
    make -C c/lib CC="emcc -I${HOME}/opt/include" prefix=${HOME}/opt install
    make -C c/bin CC="emcc -I${HOME}/opt/include" prefix=${HOME}/opt -j 6 js
    # test example:
    cd c/bin && nodejs m-nucleus.js 1000 -2 0 250 64
```

97 mandelbrot-numerics.cabal

```
name:                mandelbrot-numerics
version:             0.1.0.0
synopsis:            numeric algorithms related to the Mandelbrot set
description:         Numeric algorithms related to the Mandelbrot set:
                     ray tracing, nucleus location, bond points, etc.
homepage:            https://code.mathr.co.uk/mandelbrot-numerics
license:             GPL-3
license-file:        COPYING.md
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.Numerics,
      Mandelbrot.Numerics.BoxPeriod,
      Mandelbrot.Numerics.Child,
      Mandelbrot.Numerics.Complex,
      Mandelbrot.Numerics.DomainCoord,
      Mandelbrot.Numerics.DomainSize,
--      Mandelbrot.Numerics.ExRayIn,
--      Mandelbrot.Numerics.ExRayOut,
      Mandelbrot.Numerics.Interior,
      Mandelbrot.Numerics.Misiurewicz,
      Mandelbrot.Numerics.Nucleus,
      Mandelbrot.Numerics.Parent,
      Mandelbrot.Numerics.Progress,
      Mandelbrot.Numerics.Shape,
      Mandelbrot.Numerics.Size,
      Mandelbrot.Numerics.Wucleus
```

```
    other-extensions:
      BangPatterns,
      FlexibleContexts
    build-depends:
      base    >=4.7 && <4.14,
      safe    >=0.3 && <0.4,
      strict  >=0.3 && <0.4
    hs-source-dirs:       hs/lib
    default-language:     Haskell2010
    ghc-options:          -Wall
    ghc-prof-options:     -prof -auto-all -caf-all

test-suite mandelbrot-numerics-tests
    type:                 exitcode-stdio-1.0
    main-is:              hs/test/mandelbrot-numerics-tests.hs
    build-depends:
      base        >=4.7 && <4.12,
      QuickCheck  >=2.7 && <2.11,
      mandelbrot-numerics
    default-language:     Haskell2010
    ghc-options:          -Wall

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

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

98 README.md

```
mandelbrot-numerics
===================

Numerical algorithms related to the Mandelbrot set: ray tracing, nucleus
location, bond points, etc.


Dependencies
------------

For the C code, this may suffice on Debian-based systems:

    sudo aptitude install libmpc-dev


Build
-----

To install the C library and programs into ~/opt for example:

    make -C c/lib prefix=${HOME}/opt install
    make -C c/bin prefix=${HOME}/opt install
```

```
Run
---

To run the programs:

    LD_LIBRARY_PATH=${HOME}/opt/lib m-render # or other example program


Haskell
-------

To install the Haskell library:

    cabal install mandelbrot-numerics.cabal


Legal
-----

mandelbrot-numerics -- numerical algorithms related to the Mandelbrot set
Copyright (C) 2015-2018  Claude Heiland-Allen <claude@mathr.co.uk>

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/>.
```

99 Setup.hs

```
import Distribution.Simple
main = defaultMain
```