guide2

COLLABORATORS

	TITLE : guide2	

ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY		February 2, 2023	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME

Contents

Chapter 1

guide2

1.1 Summary of Fractal Types

```
                For detailed descriptions, select a hot-link below, see
                Fractal Types
                ,
or use <F2> from the fractal type selection screen.

SUMMARY OF FRACTAL TYPES


                barnsley
                      z(0) = pixel;
      z(n+1) = (z-1)*c if real(z) >= 0, else
      z(n+1) = (z+1)*c
    Two parameters: real and imaginary parts of c

                barnsley
                      z(0) = pixel;
      if real(z(n)) * imag(c) + real(c) * imag(z((n)) >= 0
   z(n+1) = (z(n)-1)*c
      else
   z(n+1) = (z(n)+1)*c
    Two parameters: real and imaginary parts of c

                barnsley
                      z(0) = pixel;
      if real(z(n) > 0 then z(n+1) = (real(z(n))^2 - imag(z(n))^2 - 1)
   + i * (2*real(z((n)) * imag(z((n))) else
      z(n+1) = (real(z(n))^2 - imag(z(n))^2 - 1 + real(c) * real(z(n))
       + i * (2*real(z((n)) * imag(z((n)) + imag(c) * real(z(n))
    Two parameters: real and imaginary parts of c.

                barnsley
                      z(0) = c = pixel;
      if real(z) >= 0 then
  z(n+1) = (z-1)*c
      else
  z(n+1) = (z+1)*c.
    Parameters are perturbations of z(0)
```

```
              barnsley
                    z(0) = c = pixel;
    if real(z)*imag(c) + real(c)*imag(z) >= 0
z(n+1) = (z-1)*c
    else
z(n+1) = (z+1)*c
  Parameters are perturbations of z(0)

              barnsley
                    z(0) = c = pixel;
    if real(z(n) > 0 then z(n+1) = (real(z(n))^2 - imag(z(n))^2 - 1)
 + i * (2*real(z((n)) * imag(z((n))) else
    z(n+1) = (real(z(n))^2 - imag(z(n))^2 - 1 + real(c) * real(z(n))
 + i * (2*real(z((n)) * imag(z((n)) + imag(c) * real(z(n))
  Parameters are perturbations of z(0)


              bifurcation
                  Pictorial representation of a population growth model.
    Let P = new population, p = oldpopulation, r = growth rate
    The model is: P = p + r*fn(p)*(1-fn(p)).
  Three parameters: Filter Cycles, Seed Population, and Function.

              bif+sinpi
                  Bifurcation variation: model is: P = p + r*fn(PI*p).
  Three parameters: Filter Cycles, Seed Population, and Function.

              bif=sinpi
                  Bifurcation variation: model is: P = r*fn(PI*p).
  Three parameters: Filter Cycles, Seed Population, and Function.

              biflambda
                  Bifurcation variation: model is: P = r*fn(p)*(1-fn(p)).
  Three parameters: Filter Cycles, Seed Population, and Function.

              bifstewart
                  Bifurcation variation: model is: P = (r*fn(p)*fn(p)) - 1.
  Three parameters: Filter Cycles, Seed Population, and Function.

              bifmay
                  Bifurcation variation: model is: P = r*p / ((1+p)^beta).
  Three parameters: Filter Cycles, Seed Population, and Beta.



              cellular
                  One-dimensional cellular automata or line automata.  The type  ↩
                     of CA
  is given by kr, where k is the number of different states of the
  automata and r is the radius of the neighborhood.  The next generation
  is determined by the sum of the neighborhood and the specified rule.
  Four parameters: Initial String, Rule, Type, and Starting Row Number.
  For Type = 21, 31, 41, 51, 61, 22, 32, 42, 23, 33, 24, 25, 26, 27
      Rule =  4,  7, 10, 13, 16,  6, 11, 16,  8, 15, 10, 12, 14, 16 digits


              circle
```

```
                Circle pattern by John Connett
  x + iy = pixel
  z = a*(x^2 + y^2)
  c = integer part of z
  color = c modulo(number of colors)


            cmplxmarksjul
                A generalization of the marksjulia fractal.
  z(0) = pixel;
  z(n+1) = (c^exp)*z(n)^2 + c.
Four parameters: real and imaginary parts of c and exp.



            cmplxmarksmand
                A generalization of the marksmandel fractal.
  z(0) = c = pixel;
  z(n+1) = (c^exp)*z(n)^2 + c.
Four parameters: real and imaginary parts of
perturbation of z(0) and exp.


            complexnewton, complexbasin
                Newton fractal types extended to complex degrees.  ↩
                   Complexnewton
colors pixels according to the number of iterations required to
escape to a root. Complexbasin colors pixels according to which
root captures the orbit. The equation is based on the newton
formula for solving the equation z^p = r
  z(0) = pixel;
  z(n+1) = ((p - 1) * z(n)^p + r)/(p * z(n)^(p - 1)).
Four parameters: real & imaginary parts of degree p and root r


            diffusion
                Diffusion Limited Aggregation.  Randomly moving points
accumulate.  Two parameters: border width (default 10), type



            dynamic
                Time-discrete dynamic system.
  x(0) = y(0) = start position.
  y(n+1) = y(n) + f( x(n) )
  x(n+1) = x(n) - f( y(n) )
  f(k) = sin(k + a*fn1(b*k))
For implicit Euler approximation: x(n+1) = x(n) - f( y(n+1) )
Five parameters: start position step, dt, a, b, and the function fn1.


            fn+fn(pix)
                  c = z(0) = pixel;
  z(n+1) = fn1(z) + p*fn2(c)
Six parameters: real and imaginary parts of the perturbation
of z(0) and factor p, and the functions fn1, and fn2.
```

```
            fn(z*z)
                  z(0) = pixel;
  z(n+1) = fn(z(n)*z(n))
One parameter: the function fn.


            fn*fn
                  z(0) = pixel; z(n+1) = fn1(n)*fn2(n)
Two parameters: the functions fn1 and fn2.


            fn*z+z
                  z(0) = pixel; z(n+1) = p1*fn(z(n))*z(n) + p2*z(n)
Five parameters: the real and imaginary components of
p1 and p2, and the function fn.


            fn+fn
                  z(0) = pixel;
  z(n+1) = p1*fn1(z(n))+p2*fn2(z(n))
Six parameters: The real and imaginary components of
p1 and p2, and the functions fn1 and fn2.


            formula
                Formula interpreter - write your own formulas as text files!


            frothybasin
                Pixel color is determined by which attractor captures the  ↩
                   orbit.  The
shade of color is determined by the number of iterations required to
capture the orbit.
  z(0) = pixel;  z(n+1) = z(n)^2 - c*conj(z(n))
  where c = 1 + ai,  and  a = 1.02871376822...


            gingerbread
                Orbit in two dimensions defined by:
  x(n+1) = 1 - y(n) + |x(n)|
  y(n+1) = x(n)
Two parameters: initial values of x(0) and y(0).


            halley
                  Halley map for the function: F = z(z^a - 1) = 0
  z(0) = pixel;
  z(n+1) = z(n) - R * F / [F' - (F" * F / 2 * F')]
  bailout when: abs(mod(z(n+1)) - mod(z(n)) < epsilon
Three parameters: order of z (a), relaxation coefficient (R),
                  small number for bailout (epsilon).


            henon
```

```
                Orbit in two dimensions defined by:
  x(n+1) = 1 + y(n) - a*x(n)*x(n)
  y(n+1) = b*x(n)
Two parameters: a and b


            hopalong
                Hopalong attractor by Barry Martin - orbit in two dimensions.
  z(0) = y(0) = 0;
  x(n+1) = y(n) - sign(x(n))*sqrt(abs(b*x(n)-c))
  y(n+1) = a - x(n)
Parameters are a, b, and c.


            hypercomplex
                HyperComplex Mandelbrot set.
  h(0)   = (0,0,0,0)
  h(n+1) = fn(h(n)) + C.
  where "fn" is sin, cos, log, sqr etc.
Two parameters: cj, ck
C = (xpixel,ypixel,cj,ck)



            hypercomplexj
                HyperComplex Julia set.
  h(0)   = (xpixel,ypixel,zj,zk)
  h(n+1) = fn(h(n)) + C.
  where "fn" is sin, cos, log, sqr etc.
Six parameters: c1, ci, cj, ck
C = (c1,ci,cj,ck)


            icon, icon3d
                Orbit in three dimensions defined by:
  p = lambda + alpha * magnitude + beta * (x(n)*zreal - y(n)*zimag)
  x(n+1) = p * x(n) + gamma * zreal - omega * y(n)
  y(n+1) = p * y(n) - gamma * zimag + omega * x(n)
  (3D version uses magnitude for z)
  Parameters:  Lambda, Alpha, Beta, Gamma, Omega, and Degree


            IFS
                Barnsley IFS (Iterated Function System) fractals. Apply
contractive affine mappings.



            julfn+exp
                A generalized Clifford Pickover fractal.
  z(0) = pixel;
  z(n+1) = fn(z(n)) + e^z(n) + c.
Three parameters: real & imaginary parts of c, and fn


            julfn+zsqrd
                  z(0) = pixel;
```

```
  z(n+1) = fn(z(n)) + z(n)^2 + c
Three parameters: real & imaginary parts of c, and fn


            julia
                Classic Julia set fractal.
  z(0) = pixel; z(n+1) = z(n)^2 + c.
Two parameters: real and imaginary parts of c.



            julia_inverse
                Inverse Julia function - "orbit" traces Julia set in two  ↩
                   dimensions.
  z(0) = a point on the Julia Set boundary; z(n+1) = +- sqrt(z(n) - c)
Parameters: Real and Imaginary parts of c
       Maximum Hits per Pixel (similar to max iters)
       Breadth First, Depth First or Random Walk Tree Traversal
       Left or Right First Branching (in Depth First mode only)
    Try each traversal method, keeping everything else the same.
    Notice the differences in the way the image evolves.  Start with
    a fairly low Maximum Hit limit, then increase it.  The hit limit
    cannot be higher than the maximum colors in your video mode.


            julia(fn||fn)
                  z(0) = pixel;
  if modulus(z(n)) < shift value, then
     z(n+1) = fn1(z(n)) + c,
  else
     z(n+1) = fn2(z(n)) + c.
Five parameters: real, imaginary portions of c, shift value,
                 fn1 and fn2.



            julia4
                Fourth-power Julia set fractals, a special case
of julzpower kept for speed.
  z(0) = pixel;
  z(n+1) = z(n)^4 + c.
Two parameters: real and imaginary parts of c.


            julibrot
                'Julibrot' 4-dimensional fractals.


            julzpower
                  z(0) = pixel;
  z(n+1) = z(n)^m + c.
Three parameters: real & imaginary parts of c, exponent m


            julzzpwr
                  z(0) = pixel;
  z(n+1) = z(n)^z(n) + z(n)^m + c.
```

```
Three parameters: real & imaginary parts of c, exponent m



            kamtorus, kamtorus3d
                Series of orbits superimposed.
3d version has 'orbit' the z dimension.
  x(0) = y(0) = orbit/3;
  x(n+1) = x(n)*cos(a) + (x(n)*x(n)-y(n))*sin(a)
  y(n+1) = x(n)*sin(a) - (x(n)*x(n)-y(n))*cos(a)
After each orbit, 'orbit' is incremented by a step size.
Parameters: a, step size, stop value for 'orbit', and
points per orbit.


            lambda
                Classic Lambda fractal. 'Julia' variant of Mandellambda.
  z(0) = pixel;
  z(n+1) = lambda*z(n)*(1 - z(n)).
Two parameters: real and imaginary parts of lambda.


            lambdafn
                  z(0) = pixel;
  z(n+1) = lambda * fn(z(n)).
Three parameters: real, imag portions of lambda, and fn



            lambda(fn||fn)
                  z(0) = pixel;
  if modulus(z(n)) < shift value, then
     z(n+1) = lambda * fn1(z(n)),
  else
     z(n+1) = lambda * fn2(z(n)).
Five parameters: real, imaginary portions of lambda, shift value,
                 fn1 and fn2.


            lorenz, lorenz3d
                Lorenz two lobe attractor - orbit in three dimensions.
In 2d the x and y components are projected to form the image.
  z(0) = y(0) = z(0) = 1;
  x(n+1) = x(n) + (-a*x(n)*dt) + (   a*y(n)*dt)
  y(n+1) = y(n) + ( b*x(n)*dt) - (     y(n)*dt) - (z(n)*x(n)*dt)
  z(n+1) = z(n) + (-c*z(n)*dt) + (x(n)*y(n)*dt)
Parameters are dt, a, b, and c.



            lorenz3d1
                Lorenz one lobe attractor - orbit in three dimensions.
The original formulas were developed by Rick Miranda and Emily Stone.
  z(0) = y(0) = z(0) = 1; norm = sqrt(x(n)^2 + y(n)^2)
  x(n+1) = x(n) + (-a*dt-dt)*x(n) + (a*dt-b*dt)*y(n)
     + (dt-a*dt)*norm + y(n)*dt*z(n)
  y(n+1) = y(n) + (b*dt-a*dt)*x(n) - (a*dt+dt)*y(n)
```

```
     + (b*dt+a*dt)*norm - x(n)*dt*z(n) - norm*z(n)*dt
  z(n+1) = z(n) +(y(n)*dt/2) - c*dt*z(n)
Parameters are dt, a, b, and c.
```

```
            lorenz3d3
                Lorenz three lobe attractor - orbit in three dimensions.
The original formulas were developed by Rick Miranda and Emily Stone.
  z(0) = y(0) = z(0) = 1; norm = sqrt(x(n)^2 + y(n)^2)
  x(n+1) = x(n) +(-(a*dt+dt)*x(n) + (a*dt-b*dt+z(n)*dt)*y(n))/3
      + ((dt-a*dt)*(x(n)^2-y(n)^2)
      + 2*(b*dt+a*dt-z(n)*dt)*x(n)*y(n))/(3*norm)
  y(n+1) = y(n) +((b*dt-a*dt-z(n)*dt)*x(n) - (a*dt+dt)*y(n))/3
      + (2*(a*dt-dt)*x(n)*y(n)
      + (b*dt+a*dt-z(n)*dt)*(x(n)^2-y(n)^2))/(3*norm)
  z(n+1) = z(n) +(3*x(n)*dt*x(n)*y(n)-y(n)*dt*y(n)^2)/2 - c*dt*z(n)
Parameters are dt, a, b, and c.
```

```
            lorenz3d4
                Lorenz four lobe attractor - orbit in three dimensions.
The original formulas were developed by Rick Miranda and Emily Stone.
  z(0) = y(0) = z(0) = 1;
  x(n+1) = x(n) +(-a*dt*x(n)^3
     + (2*a*dt+b*dt-z(n)*dt)*x(n)^2*y(n) + (a*dt-2*dt)*x(n)*y(n)^2
     + (z(n)*dt-b*dt)*y(n)^3) / (2 * (x(n)^2+y(n)^2))
  y(n+1) = y(n) +((b*dt-z(n)*dt)*x(n)^3 + (a*dt-2*dt)*x(n)^2*y(n)
     + (-2*a*dt-b*dt+z(n)*dt)*x(n)*y(n)^2
     - a*dt*y(n)^3) / (2 * (x(n)^2+y(n)^2))
  z(n+1) = z(n) +(2*x(n)*dt*x(n)^2*y(n) - 2*x(n)*dt*y(n)^3 - c*dt*z(n))
Parameters are dt, a, b, and c.
```

```
            lsystem
                Using a turtle-graphics control language and starting with
an initial axiom string, carries out string substitutions the
specified number of times (the order), and plots the resulting.
```

```
            lyapunov
                Derived from the Bifurcation fractal, the Lyapunov plots the ↩
                   Lyapunov
Exponent for a population model where the Growth parameter varies between
two values in a periodic manner.
```

```
            magnet1j
                  z(0) = pixel;
[  z(n)^2 + (c-1)  ] 2
  z(n+1) =  | ---------------- |
[  2*z(n) + (c-2)  ]
Parameters: the real and imaginary parts of c
```

```
            magnet1m
```

```
                  z(0) = 0; c = pixel;
[  z(n)^2 + (c-1)  ] 2
  z(n+1) =  | ---------------- |
[  2*z(n) + (c-2)  ]
Parameters: the real & imaginary parts of perturbation of z(0)


            magnet2j
                  z(0) = pixel;
[  z(n)^3 + 3*(C-1)*z(n) + (C-1)*(C-2)         ] 2
  z(n+1) =  |  ---------------------------------------- |
[  3*(z(n)^2) + 3*(C-2)*z(n) + (C-1)*(C-2) + 1 ]
Parameters: the real and imaginary parts of c


            magnet2m
                  z(0) = 0; c = pixel;
[  z(n)^3 + 3*(C-1)*z(n) + (C-1)*(C-2)         ] 2
  z(n+1) =  |  ---------------------------------------- |
[  3*(z(n)^2) + 3*(C-2)*z(n) + (C-1)*(C-2) + 1 ]
Parameters: the real and imaginary parts of perturbation of z(0)


            mandel
                Classic Mandelbrot set fractal.
  z(0) = c = pixel;
  z(n+1) = z(n)^2 + c.
Two parameters: real & imaginary perturbations of z(0)


            mandel(fn||fn)
                  c = pixel;
  z(0) = p1
  if modulus(z(n)) < shift value, then
     z(n+1) = fn1(z(n)) + c,
  else
     z(n+1) = fn2(z(n)) + c.
Five parameters: real, imaginary portions of p1, shift value,
                 fn1 and fn2.



            mandelcloud
                Displays orbits of Mandelbrot set:
  z(0) = c = pixel;
  z(n+1) = z(n)^2 + c.
One parameter: number of intervals


            mandel4
                Special case of mandelzpower kept for speed.
  z(0) = c = pixel;
  z(n+1) = z(n)^4 + c.
Parameters: real & imaginary perturbations of z(0)


            mandelfn
```

```
                z(0) = c = pixel;
  z(n+1) = c*fn(z(n)).
Parameters: real & imaginary perturbations of z(0), and fn


          manlam(fn||fn)
                c = pixel;
  z(0) = p1
  if modulus(z(n)) < shift value, then
     z(n+1) = fn1(z(n)) * c, else
     z(n+1) = fn2(z(n)) * c.
Five parameters: real, imaginary parts of p1, shift value, fn1, fn2.


          Martin
              Attractor fractal by Barry Martin - orbit in two dimensions.
  z(0) = y(0) = 0;
  x(n+1) = y(n) - sin(x(n))
  y(n+1) = a - x(n)
Parameter is a (try a value near pi)


          mandellambda
                z(0) = .5; lambda = pixel;
  z(n+1) = lambda*z(n)*(1 - z(n)).
Parameters: real & imaginary perturbations of z(0)


          mandphoenix
                z(0) = p1, y(0) = 0;
  For degree of Z = 0:
    z(n+1) = z(n)^2 + pixel.x + (pixel.y)y(n), y(n+1) = z(n)
  For degree of Z >= 2:
    z(n+1) = z(n)^degree + pz(n)^(degree-1) + qy(n), y(n+1) = z(n)
  For degree of Z <= -3:
    z(n+1) = z(n)^|degree| + pz(n)^(|degree|-2) + qy(n), y(n+1) = z(n)
Three parameters: real part of z(0), imaginary part of z(0), and the
  degree of Z.


          manfn+exp
              'Mandelbrot-Equivalent' for the julfn+exp fractal.
  z(0) = c = pixel;
  z(n+1) = fn(z(n)) + e^z(n) + C.
Parameters: real & imaginary perturbations of z(0), and fn


          manfn+zsqrd
              'Mandelbrot-Equivalent' for the Julfn+zsqrd fractal.
  z(0) = c = pixel;
  z(n+1) = fn(z(n)) + z(n)^2 + c.
Parameters: real & imaginary perturbations of z(0), and fn
```

```
            manowar
                  c = z1(0) = z(0) = pixel;
  z(n+1) = z(n)^2 + z1(n) + c;
  z1(n+1) = z(n);
Parameters: real & imaginary perturbations of z(0)


            manowar
                  z1(0) = z(0) = pixel;
  z(n+1) = z(n)^2 + z1(n) + c;
  z1(n+1) = z(n);
Parameters: real & imaginary perturbations of c



            manzpower
                'Mandelbrot-Equivalent' for julzpower.
  z(0) = c = pixel;
  z(n+1) = z(n)^exp + c; try exp = e = 2.71828...
Parameters: real & imaginary perturbations of z(0), real &
imaginary parts of exponent exp.


            manzzpwr
                'Mandelbrot-Equivalent' for the julzzpwr fractal.
  z(0) = c = pixel
  z(n+1) = z(n)^z(n) + z(n)^exp + C.
Parameters: real & imaginary perturbations of z(0), and exponent


            marksjulia
                A variant of the julia-lambda fractal.
  z(0) = pixel;
  z(n+1) = (c^exp)*z(n)^2 + c.
Parameters: real & imaginary parts of c, and exponent



            marksmandel
                A variant of the mandel-lambda fractal.
  z(0) = c = pixel;
  z(n+1) = (c^exp)*z(n)^2 + c.
Parameters: real & imaginary perturbations of z(0), and exponent


            marksmandelpwr
                The marksmandelpwr formula type generalized (it previously
had fn=sqr hard coded).
  z(0) = pixel, c = z(0) ^ (z(0) - 1):
  z(n+1) = c * fn(z(n)) + pixel,
Parameters: real and imaginary perturbations of z(0), and fn


            newtbasin
                Based on the Newton formula for finding the roots of z^p - 1.
Pixels are colored according to which root captures the orbit.
  z(0) = pixel;
```

```
  z(n+1) = ((p-1)*z(n)^p + 1)/(p*z(n)^(p - 1)).
Two parameters: the polynomial degree p, and a flag to turn
on color stripes to show alternate iterations.


            newton
                Based on the Newton formula for finding the roots of z^p - 1.
Pixels are colored according to the iteration when the orbit
is captured by a root.
  z(0) = pixel;
  z(n+1) = ((p-1)*z(n)^p + 1)/(p*z(n)^(p - 1)).
One parameter: the polynomial degree p.


            phoenix
                  z(0) = pixel, y(0) = 0;
  For degree of Z = 0: z(n+1) = z(n)^2 + p + qy(n), y(n+1) = z(n)
  For degree of Z >= 2:
    z(n+1) = z(n)^degree + pz(n)^(degree-1) + qy(n), y(n+1) = z(n)
  For degree of Z <= -3:
    z(n+1) = z(n)^|degree| + pz(n)^(|degree|-2) + qy(n), y(n+1) = z(n)
Three parameters: real p, real q, and the degree of Z.


            pickover
                Orbit in three dimensions defined by:
  x(n+1) = sin(a*y(n)) - z(n)*cos(b*x(n))
  y(n+1) = z(n)*sin(c*x(n)) - cos(d*y(n))
  z(n+1) = sin(x(n))
Parameters: a, b, c, and d.


            plasma
                Random, cloud-like formations.  Requires 4 or more colors.
A recursive algorithm repeatedly subdivides the screen and
colors pixels according to an average of surrounding pixels
and a random color, less random as the grid size decreases.
Four parameters: 'graininess' (.5 to 50, default = 2), old/new
algorithm, seed value used, 16-bit out output selection.


            popcorn
                The orbits in two dimensions defined by:
  x(0) = xpixel, y(0) = ypixel;
  x(n+1) = x(n) - h*sin(y(n) + tan(3*y(n))
  y(n+1) = y(n) - h*sin(x(n) + tan(3*x(n))
are plotted for each screen pixel and superimposed.
One parameter: step size h.


            popcornjul
                Conventional Julia using the popcorn formula:
  x(0) = xpixel, y(0) = ypixel;
  x(n+1) = x(n) - h*sin(y(n) + tan(3*y(n))
  y(n+1) = y(n) - h*sin(x(n) + tan(3*x(n))
```

```
One parameter: step size h.


            quatjul
                Quaternion Julia set.
  q(0)   = (xpixel,ypixel,zj,zk)
  q(n+1) = q(n)*q(n) + c.
Four parameters: c, ci, cj, ck
c = (c1,ci,cj,ck)


            quat
                Quaternion Mandelbrot set.
  q(0)   = (0,0,0,0)
  q(n+1) = q(n)*q(n) + c.
Two parameters: cj,ck
c = (xpixel,ypixel,cj,ck)



            rossler3D
                Orbit in three dimensions defined by:
  x(0) = y(0) = z(0) = 1;
  x(n+1) = x(n) - y(n)*dt -   z(n)*dt
  y(n+1) = y(n) + x(n)*dt + a*y(n)*dt
  z(n+1) = z(n) + b*dt + x(n)*z(n)*dt - c*z(n)*dt
Parameters are dt, a, b, and c.


            sierpinski
                Sierpinski gasket - Julia set producing a 'Swiss cheese  ↩
                   triangle'
  z(n+1) = (2*x,2*y-1) if y > .5;
else (2*x-1,2*y) if x > .5;
else (2*x,2*y)
No parameters.


            spider
                  c(0) = z(0) = pixel;
  z(n+1) = z(n)^2 + c(n);
  c(n+1) = c(n)/2 + z(n+1)
Parameters: real & imaginary perturbation of z(0)



            sqr(1/fn)
                  z(0) = pixel;
  z(n+1) = (1/fn(z(n))^2
One parameter: the function fn.


            sqr(fn)
                  z(0) = pixel;
  z(n+1) = fn(z(n))^2
One parameter: the function fn.
```

```
test
    'test' point letting us (and you!) easily add fractal types via
    the c module testpt.c.  Default set up is a mandelbrot fractal.
    Four parameters: user hooks (not used by default testpt.c).


                tetrate
                      z(0) = c = pixel;
      z(n+1) = c^z(n)
    Parameters: real & imaginary perturbation of z(0)



                tim's_error
                    A serendipitous coding error in marksmandelpwr brings to life
    an ancient pterodactyl!  (Try setting fn to sqr.)
      z(0) = pixel, c = z(0) ^ (z(0) - 1):
      tmp = fn(z(n))
      real(tmp) = real(tmp) * real(c) - imag(tmp) * imag(c);
      imag(tmp) = real(tmp) * imag(c) - imag(tmp) * real(c);
      z(n+1) = tmp + pixel;
    Parameters: real & imaginary perturbations of z(0) and function fn


                unity
                      z(0) = pixel;
      x = real(z(n)), y = imag(z(n))
      One = x^2 + y^2;
      y = (2 - One) * x;
      x = (2 - One) * y;
      z(n+1) = x + i*y
    No parameters.
```

1.2 Fractal Types

```
                A list of the fractal types and their mathematics can be found in ↩
                   the
Summary of Fractal Types.  Some notes about how Fractint calculates
them are in "A Little Code" in Fractals and the PC.

Fractint starts by default with the Mandelbrot set. You can change that by
using the command-line argument "TYPE=" followed by one of the
fractal type names, or by using the <T> command and
selecting the type - if parameters are needed, you will be prompted for
them.

In the text that follows, due to the limitations of the ASCII character
set, "a*b" means "a times b", and "a^b" means "a to the power b".

Select a fractal type:


                 The Mandelbrot Set

                 Julia Sets
```

Inverse Julias

Newton domains of attraction

Newton

Complex Newton

Lambda Sets

Mandellambda Sets

Plasma Clouds

Lambdafn

Mandelfn

Barnsley Mandelbrot/Julia Sets

Barnsley IFS Fractals

Sierpinski Gasket

Quartic Mandelbrot/Julia

Distance Estimator

Pickover Mandelbrot/Julia Types

Pickover Popcorn

Dynamic System

Quaternion

Peterson Variations

Unity

Circle

Scott Taylor / Lee Skinner Variations

Kam Torus

Bifurcation

Orbit Fractals

Lorenz Attractors

Rossler Attractors

Henon Attractors

Pickover Attractors

Martin Attractors

Gingerbreadman
Test

Formula

Julibrots

Diffusion Limited Aggregation

Magnetic Fractals

L-Systems

Lyapunov Fractals

fn||fn Fractals

Halley

Cellular Automata

Phoenix

Frothy Basins

Icon

Hypercomplex

1.3 The Mandelbrot Set

(type=mandel)

This set is the classic: the only one implemented in many plotting programs, and the source of most of the printed fractal images published in recent years. Like most of the other types in Fractint, it is simply a graph: the x (horizontal) and y (vertical) coordinate axes represent ranges of two independent quantities, with various colors used to symbolize levels of a third quantity which depends on the first two. So far, so good: basic analytic geometry.

Now things get a bit hairier. The x axis is ordinary, vanilla real numbers. The y axis is an imaginary number, i.e. a real number times i, where i is the square root of -1. Every point on the plane -- in this case, your PC's display screen -- represents a complex number of the form:

    x-coordinate + i * y-coordinate

If your math training stopped before you got to imaginary and complex numbers, this is not the place to catch up. Suffice it to say that they are just as "real" as the numbers you count fingers with (they're used

every day by electrical engineers) and they can undergo the same kinds of algebraic operations.

OK, now pick any complex number -- any point on the complex plane -- and call it C, a constant. Pick another, this time one which can vary, and call it Z. Starting with Z=0 (i.e., at the origin, where the real and imaginary axes cross), calculate the value of the expression

    Z^2 + C

Take the result, make it the new value of the variable Z, and calculate again. Take that result, make it Z, and do it again, and so on: in mathematical terms, iterate the function Z(n+1) = Z(n)^2 + C. For certain values of C, the result "levels off" after a while. For all others, it grows without limit. The Mandelbrot set you see at the start -- the solid-colored lake (blue by default), the blue circles sprouting from it, and indeed every point of that color -- is the set of all points C for which the value of Z is less than 2 after 150 iterations (150 is the default setting, changeable via the <X> options screen or "maxiter=" parameter).
All the surrounding "contours" of other colors represent points for which Z exceeds 2 after 149 iterations (the contour closest to the M-set itself), 148 iterations, (the next one out), and so on.

We actually don't test for Z exceeding 2 - we test Z squared against 4 instead because it is easier. This value (FOUR usually) is known as the "bailout" value for the calculation, because we stop iterating for the point when it is reached. The bailout value can be changed on the <Z> options screen but the default is usually best.

Some features of interest:

1. Use the <X> options screen to increase the maximum number of iterations. Notice that the boundary of the M-set becomes more and more convoluted (the technical terms are "wiggly," "squiggly," and "utterly bizarre") as the Z-values for points that were still within the set after 150 iterations turn out to exceed 2 after 200, 500, or 1200. In fact, it can be proven that the true boundary is infinitely long: detail without limit.

2. Although there appear to be isolated "islands" of blue, zoom in -- that is, plot for a smaller range of coordinates to show more detail -- and you'll see that there are fine "causeways" of blue connecting them to the main set. As you zoomed, smaller islands became visible; the same is true for them. In fact, there are no isolated points in the M-set: it is "connected" in a strict mathematical sense.

3. The upper and lower halves of the first image are symmetric (a fact that Fractint makes use of here and in some other fractal types to speed plotting). But notice that the same general features -- lobed discs, spirals, starbursts -- tend to repeat themselves (although never exactly) at smaller and smaller scales, so that it can be impossible to judge by eye the scale of a given image.

4. In a sense, the contour colors are window-dressing: mathematically, it is the properties of the M-set itself that are interesting, and no information about it would be lost if all points outside the set were assigned the same color. If you're a serious, no-nonsense type, you may want to cycle the colors just once to see the kind of silliness that other

people enjoy, and then never do it again. Go ahead. Just once, now. We trust you.

1.4 Julia Sets

(type=julia)

These sets were named for mathematician Gaston Julia, and can be generated by a simple change in the iteration process described for the

The Mandelbrot Set
. Start with a
specified value of C, "C-real + i * C-imaginary"; use as the initial value of Z "x-coordinate + i * y-coordinate"; and repeat the same iteration, Z(n+1) = Z(n)^2 + C.

There is a Julia set corresponding to every point on the complex plane -- an infinite number of Julia sets. But the most visually interesting tend to be found for the same C values where the M-set image is busiest, i.e. points just outside the boundary. Go too far inside, and the corresponding Julia set is a circle; go too far outside, and it breaks up into scattered points. In fact, all Julia sets for C within the M-set share the "connected" property of the M-set, and all those for C outside lack it.

Fractint's spacebar toggle lets you "flip" between any view of the M-set and the Julia set for the point C at the center of that screen. You can then toggle back, or zoom your way into the Julia set for a while and then return to the M-set. So if the infinite complexity of the M-set palls, remember: each of its infinite points opens up a whole new Julia set.

Historically, the Julia sets came first: it was while looking at the M-set as an "index" of all the Julia sets' origins that Mandelbrot noticed its properties.

The relationship between the
Mandelbrot
set and Julia set can
hold between
other sets as well. Many of Fractint's types are "Mandelbrot/Julia" pairs (sometimes called "M-sets" or "J-sets". All these are generated by equations that are of the form z(k+1) = f(z(k),c), where the function orbit is the sequence z(0), z(1), ..., and the variable c is a complex parameter of the equation. The value c is fixed for "Julia" sets and is equal to the first two parameters entered with the "params=Creal/Cimag" command. The initial orbit value z(0) is the complex number corresponding to the screen pixel. For Mandelbrot sets, the parameter c is the complex number corresponding to the screen pixel. The value z(0) is c plus a perturbation equal to the values of the first two parameters. See
the discussion of
Mandellambda Sets
.
This approach may or may not be the
"standard" way to create "Mandelbrot" sets out of "Julia" sets.

Some equations have additional parameters. These values are entered as the

third for fourth params= value for both Julia and Mandelbrot sets. The variables x and y refer to the real and imaginary parts of z; similarly, cx and cy are the real and imaginary parts of the parameter c and fx(z) and fy(z) are the real and imaginary parts of f(z). The variable c is sometimes called lambda for historical reasons.

NOTE: if you use the "PARAMS=" argument to warp the M-set by starting with an initial value of Z other than 0, the M-set/J-sets correspondence breaks down and the spacebar toggle no longer works.

1.5 Julia Toggle Spacebar Commands

The spacebar toggle has been enhanced for the classic Mandelbrot and Julia types. When viewing the Mandelbrot, the spacebar turns on a window mode that displays the Inverse Julia corresponding to the cursor position in a window. Pressing the spacebar then causes the regular Julia escape time fractal corresponding to the cursor position to be generated. The following keys take effect in Inverse Julia mode.

<Space> Generate the escape-time Julia Set corresponding to the cursor position. Only works if fractal is a "Mandelbrot" type.
<n> Numbers toggle - shows coordinates of the cursor on the screen. Press <n> again to turn off numbers.
<p> Enter new pixel coordinates directly
<h> Hide fractal toggle. Works only if View Windows is turned on and set for a small window (such as the default size.) Hides the fractal, allowing the orbit to take up the whole screen. Press <h> again to uncover the fractal.
<s> Saves the fractal, cursor, orbits, and numbers.
<<> or <,> Zoom inverse julia image smaller.
<>> or <.> Zoom inverse julia image larger.
<z> Restore default zoom.

The Julia Inverse window is only implemented for the classic Mandelbrot (type=mandel). For other "Mandelbrot" types <space> turns on the cursor without the Julia window, and allows you to select coordinates of the matching Julia set in a way similar to the use of the zoom box with the Mandelbrot/Julia toggle in previous Fractint versions.

1.6 Inverse Julias

(type=julia_inverse)

Pick a function, such as the familiar Z(n) = Z(n-1) squared plus C (the defining function of the Mandelbrot Set). If you pick a point Z(0) at random from the complex plane, and repeatedly apply the function to it, you get a sequence of new points called an orbit, which usually either zips out toward infinity or zooms in toward one or more "attractor" points near the middle of the plane. The set of all points that are "attracted" to infinity is called the "Basin of Attraction" of infinity. Each of the other attractors also has its own Basin of Attraction. Why is it called a Basin? Imagine a lake, and all the water in it "draining" into the

attractor. The boundary between these basins is called the Julia Set of the function.

The boundary between the basins of attraction is sort of like a repeller; all orbits move away from it, toward one of the attractors. But if we define a new function as the inverse of the old one, as for instance Z(n) = sqrt(Z(n-1) minus C), then the old attractors become repellers, and the former boundary itself becomes the attractor! Now, starting from any point, all orbits are drawn irresistibly to the Julia Set! In fact, once an orbit reaches the boundary, it will continue to hop about until it traces the entire Julia Set! This method for drawing Julia Sets is called the Inverse Iteration Method, or IIM for short.

Unfortunately, some parts of each Julia Set boundary are far more attractive to inverse orbits than others are, so that as an orbit traces out the set, it keeps coming back to these attractive parts again and again, only occasionally visiting the less attractive parts. Thus it may take an infinite length of time to draw the entire set. To hasten the process, we can keep track of how many times each pixel on our computer screen is visited by an orbit, and whenever an orbit reaches a pixel that has already been visited more than a certain number of times, we can consider that orbit finished and move on to another one. This "hit limit" thus becomes similar to the iteration limit used in the traditional escape-time fractal algorithm. This is called the Modified Inverse Iteration Method, or MIIM, and is much faster than the IIM.

Now, the inverse of Mandelbrot's classic function is a square root, and the square root actually has two solutions; one positive, one negative. Therefore at each step of each orbit of the inverse function there is a decision; whether to use the positive or the negative square root. Each one gives rise to a new point on the Julia Set, so each is a good choice. This series of choices defines a binary decision tree, each point on the Julia Set giving rise to two potential child points. There are many interesting ways to traverse a binary tree, among them Breadth first, Depth first (left or negative first), Depth first (right or positive first), and completely at random. It turns out that most traversal methods lead to the same or similar pictures, but that how the image evolves as the orbits trace it out differs wildly depending on the traversal method chosen. As far as I know, this fact is an original discovery, and this version of FRACTINT is its first publication.

Pick a Julia constant such as Z(0) = (-.74543, .11301), the popular Seahorse Julia, and try drawing it first Breadth first, then Depth first (right first), Depth first (left first), and finally with Random Walk.

Caveats: the video memory is used in the algorithm, to keep track of how many times each pixel has been visited (by changing it's color). Therefore the algorithm will not work well if you zoom in far enough that part of the Julia Set is off the screen.

Bugs: Not working with Disk Video.
Not resumeable.

The <J> key toggles between the Inverse Julia orbit and the corresponding Julia escape time fractal.

1.7 Newton domains of attraction

(type=newtbasin)

The Newton formula is an algorithm used to find the roots of polynomial equations by successive "guesses" that converge on the correct value as you feed the results of each approximation back into the formula. It works very well -- unless you are unlucky enough to pick a value that is on a line BETWEEN two actual roots. In that case, the sequence explodes into chaos, with results that diverge more and more wildly as you continue the iteration.

This fractal type shows the results for the polynomial Z^n - 1, which has n roots in the complex plane. Use the <T>ype command and enter "newtbasin" in response to the prompt. You will be asked for a parameter, the "order" of the equation (an integer from 3 through 10 -- 3 for x^3-1, 7 for x^7-1, etc.). A second parameter is a flag to turn on alternating shades showing changes in the number of iterations needed to attract an orbit. Some people like stripes and some don't, as always, Fractint gives you a choice!

The coloring of the plot shows the "basins of attraction" for each root of the polynomial -- i.e., an initial guess within any area of a given color would lead you to one of the roots. As you can see, things get a bit weird along certain radial lines or "spokes," those being the lines between actual roots. By "weird," we mean infinitely complex in the good old fractal sense. Zoom in and see for yourself.

This fractal type is symmetric about the origin, with the number of "spokes" depending on the order you select. It uses floating-point math if you have an FPU, or a somewhat slower integer algorithm if you don't have one.

See also:
Newton

1.8 Newton

(type=newton)

The generating formula here is identical to that for
newtbasin
,
but the
coloring scheme is different. Pixels are colored not according to the root that would be "converged on" if you started using Newton's formula from that point, but according to the iteration when the value is close to a root. For example, if the calculations for a particular pixel converge to the 7th root on the 23rd iteration, NEWTBASIN will color that pixel using color #7, but NEWTON will color it using color #23.

If you have a 256-color mode, use it: the effects can be much livelier than those you get with type=newtbasin, and color cycling becomes, like, downright cosmic. If your "corners" choice is symmetrical, Fractint

exploits the symmetry for faster display.

The applicable "params=" values are the same as newtbasin. Try "params=4." Other values are 3 through 10. 8 has twice the symmetry and is faster. As with newtbasin, an FPU helps.

1.9 Complex Newton

(type=complexnewton/complexbasin)

Well, hey, "Z^n - 1" is so boring when you can use "Z^a - b" where "a" and "b" are complex numbers! The new "complexnewton" and "complexbasin" fractal types are just the old
newton
and

newtbasin
fractal types with
this little added twist. When you select these fractal types, you are prompted for four values (the real and imaginary portions of "a" and "b"). If "a" has a complex portion, the fractal has a discontinuity along the negative axis - relax, we finally figured out that it's *supposed* to be there!

1.10 Lambda Sets

(type=lambda)

This type calculates the Julia set of the formula lambda*Z*(1-Z). That is, the value Z[0] is initialized with the value corresponding to each pixel position, and the formula iterated. The pixel is colored according to the iteration when the sum of the squares of the real and imaginary parts exceeds 4.

Two parameters, the real and imaginary parts of lambda, are required. Try 0 and 1 to see the classical fractal "dragon". Then try 0.2 and 1 for a lot more detail to zoom in on.

It turns out that all quadratic Julia-type sets can be calculated using just the formula z^2+c (the "classic" Julia"), so that this type is redundant, but we include it for reason of it's prominence in the history of fractals.

1.11 Mandellambda Sets

(type=mandellambda)

This type is the "Mandelbrot equivalent of the lambda set.
A comment is
in order here. Almost all the Fractint "Mandelbrot" sets are created from

orbits generated using formulas like z(n+1) = f(z(n),C), with z(0) and C initialized to the complex value corresponding to the current pixel. Our reasoning was that "Mandelbrots" are maps of the corresponding "Julias". Using this scheme each pixel of a "Mandelbrot" is colored the same as the Julia set corresponding to that pixel. However, Kevin Allen informs us that the MANDELLAMBDA set appears in the literature with z(0) initialized to a critical point (a point where the derivative of the formula is zero), which in this case happens to be the point (.5,0). Since Kevin knows more about Dr. Mandelbrot than we do, and Dr. Mandelbrot knows more about fractals than we do, we defer! Starting with version 14 Fractint calculates MANDELAMBDA Dr. Mandelbrot's way instead of our way. But ALL THE OTHER "Mandelbrot" sets in Fractint are still calculated OUR way! (Fortunately for us, for the classic Mandelbrot Set these two methods are the same!)

Well now, folks, apart from questions of faithfulness to fractals named in the literature (which we DO take seriously!), if a formula makes a beautiful fractal, it is not wrong. In fact some of the best fractals in Fractint are the results of mistakes! Nevertheless, thanks to Kevin for keeping us accurate!

(See description of "initorbit=" command in Image Calculation Parameters for a way to experiment with different orbit intializations).

1.12 Circle

(type=circle)

This fractal types is from A. K. Dewdney's "Computer Recreations" column in "Scientific American". It is attributed to John Connett of the University of Minnesota.

(Don't tell anyone, but this fractal type is not really a fractal!)

Fascinating Moire patterns can be formed by calculating x^2 + y^2 for each pixel in a piece of the complex plane. After multiplication by a magnification factor (the parameter), the number is truncated to an integer and mapped to a color via color = value modulo (number of colors). That is, the integer is divided by the number of colors, and the remainder is the color index value used. The resulting image is not a fractal because all detail is lost after zooming in too far. Try it with different resolution video modes - the results may surprise you!

1.13 Plasma Clouds

(type=plasma)

Plasma clouds ARE real live fractals, even though we didn't know it at first. They are generated by a recursive algorithm that randomly picks colors of the corner of a rectangle, and then continues recursively quartering previous rectangles. Random colors are averaged with those of the outer rectangles so that small neighborhoods do not show much change,

for a smoothed-out, cloud-like effect. The more colors your video mode supports, the better. The result, believe it or not, is a fractal landscape viewed as a contour map, with colors indicating constant elevation. To see this, save and view with the <3> command (see 3D Images) and your "cloud" will be converted to a mountain!

You've GOT to try color cycling on these (hit "+" or "-"). If you haven't been hypnotized by the drawing process, the writhing colors will do it for sure. We have now implemented subliminal messages to exploit the user's vulnerable state; their content varies with your bank balance, politics, gender, accessibility to a Fractint programmer, and so on. A free copy of Microsoft C to the first person who spots them.

This type accepts four parameters.

The first determines how abruptly the colors change. A value of .5 yields bland clouds, while 50 yields very grainy ones. The default value is 2.

The second determines whether to use the original algorithm (0) or a modified one (1). The new one gives the same type of images but draws the dots in a different order. It will let you see what the final image will look like much sooner than the old one.

The third determines whether to use a new seed for generating the next plasma cloud (0) or to use the previous seed (1).

The fourth parameter turns on 16-bit .POT output which provides much smoother height gradations. This is especially useful for creating mountain landscapes when using the plasma output with a ray tracer such as POV-Ray.

With parameter three set to 1, the next plasma cloud generated will be identical to the previous but at whatever new resolution is desired.

Zooming is ignored, as each plasma-cloud screen is generated randomly.

The random number seed used for each plasma image is displayed on the <tab> information screen, and can be entered with the command line parameter "rseed=" to recreate a particular image.

The algorithm is based on the Pascal program distributed by Bret Mulvey as PLASMA.ARC. We have ported it to C and integrated it with Fractint's graphics and animation facilities. This implementation does not use floating-point math. The algorithm was modified starting with version 18 so that the plasma effect is independent of screen resolution.

Saved plasma-cloud screens are EXCELLENT starting images for fractal "landscapes" created with the 3D commands.

1.14 Lambdafn

(type=lambdafn)

Function=[sin|cos|sinh|cosh|exp|log|sqr|...]) is specified with this type. Prior to version 14, these types were lambdasine, lambdacos, lambdasinh, lambdacos, and lambdaexp. Where we say "lambdasine" or some such below, the good reader knows we mean "lambdafn with function=sin".)

These types calculate the Julia set of the formula lambda*fn(Z), for various values of the function "fn", where lambda and Z are both complex. Two values, the real and imaginary parts of lambda, should be given in the "params=" option. For the feathery, nested spirals of LambdaSines and the frost-on-glass patterns of LambdaCosines, make the real part = 1, and try values for the imaginary part ranging from 0.1 to 0.4 (hint: values near 0.4 have the best patterns). In these ranges the Julia set "explodes". For the tongues and blobs of LambdaExponents, try a real part of 0.379 and an imaginary part of 0.479.

A coprocessor used to be almost mandatory: each LambdaSine/Cosine iteration calculates a hyperbolic sine, hyperbolic cosine, a sine, and a cosine (the LambdaExponent iteration "only" requires an exponent, sine, and cosine operation)! However, Fractint now computes these transcendental functions with fast integer math. In a few cases the fast math is less accurate, so we have kept the old slow floating point code. To use the old code, invoke with the float=yes option, and, if you DON'T have a coprocessor, go on a LONG vacation!

1.15 Halley

(type=halley)

The Halley map is an algorithm used to find the roots of polynomial equations by successive "guesses" that converge on the correct value as you feed the results of each approximation back into the formula. It works very well -- unless you are unlucky enough to pick a value that is on a line BETWEEN two actual roots. In that case, the sequence explodes into chaos, with results that diverge more and more wildly as you continue the iteration.

This fractal type shows the results for the polynomial Z(Z^a - 1), which has a+1 roots in the complex plane. Use the <T>ype command and enter "halley" in response to the prompt. You will be asked for a parameter, the "order" of the equation (an integer from 2 through 10 -- 2 for Z(Z^2 - 1), 7 for Z(Z^7 - 1), etc.). A second parameter is the relaxation coefficient, and is used to control the convergence stability. A number greater than one increases the chaotic behavior and a number less than one decreases the chaotic behavior. The third parameter is the value used to determine when the formula has converged. The test for convergence is ||Z(n+1)|^2 - |Z(n)|^2| < epsilon. This convergence test produces the whisker-like projections which generally point to a root.

1.16 Phoenix

(type=phoenix, mandphoenix)

The phoenix type defaults to the original phoenix curve discovered by
Shigehiro Ushiki, "Phoenix", IEEE Transactions on Circuits and Systems,
Vol. 35, No. 7, July 1988, pp. 788-789. These images do not have the
X and Y axis swapped as is normal for this type.

The mandphoenix type is the corresponding Mandelbrot set image of the
phoenix type. The spacebar toggles between the two as long as the
mandphoenix type has an initial Z(0) of (0,0). The mandphoenix is not
an effective index to the phoenix type, so explore the wild blue yonder.

To reproduce the Mandelbrot set image of the phoenix type as shown in
Stevens' book, "Fractal Programming in C", set initorbit=0/0 on the
command line or with the <g> key. The colors need to be rotated one
position because Stevens uses the values from the previous calculation
instead of the current calculation to determine when to bailout.

1.17 fn||fn Fractals

(type=lambda(fn||fn), manlam(fn||fn), julia(fn||fn), mandel(fn||fn))

Two functions=[sin|cos|sinh|cosh|exp|log|sqr|...]) are specified with
these types. The two functions are alternately used in the calculation
based on a comparison between the modulus of the current Z and the
shift value. The first function is used if the modulus of Z is less
than the shift value and the second function is used otherwise.

The lambda(fn||fn) type calculates the Julia set of the formula
lambda*fn(Z), for various values of the function "fn", where lambda
and Z are both complex. Two values, the real and imaginary parts of
lambda, should be given in the "params=" option. The third value is
the shift value. The space bar will generate the corresponding
"psuedo Mandelbrot" set, manlam(fn||fn).

The manlam(fn||fn) type calculates the "psuedo Mandelbrot" set of the
formula fn(Z)*C, for various values of the function "fn", where C
and Z are both complex. Two values, the real and imaginary parts of
Z(0), should be given in the "params=" option. The third value is
the shift value. The space bar will generate the corresponding
julia set, lamda(fn||fn).

The julia(fn||fn) type calculates the Julia set of the formula
fn(Z)+C, for various values of the function "fn", where C
and Z are both complex. Two values, the real and imaginary parts of
C, should be given in the "params=" option. The third value is
the shift value. The space bar will generate the corresponding
mandelbrot set, mandel(fn||fn).

The mandel(fn||fn) type calculates the Mandelbrot set of the formula
fn(Z)+C, for various values of the function "fn", where C
and Z are both complex. Two values, the real and imaginary parts of
Z(0), should be given in the "params=" option. The third value is
the shift value. The space bar will generate the corresponding
julia set, julia(fn||fn).

1.18 Mandelfn

(type=mandelfn)

Function=[sin|cos|sinh|cosh|exp|log|sqr|...]) is specified with this type. Prior to version 14, these types were mandelsine, mandelcos, mandelsinh, mandelcos, and mandelexp. Same comment about our lapses into the old terminology as above!

These are "pseudo-Mandelbrot" mappings for the LambdaFn
Julia functions.
They map to their corresponding Julia sets via the spacebar command in exactly the same fashion as the original M/J sets. In general, they are interesting mainly because of that property (the function=exp set in particular is rather boring). Generate the appropriate "Mandelfn" set, zoom on a likely spot where the colors are changing rapidly, and hit the spacebar key to plot the Julia set for that particular point.

Try "FRACTINT TYPE=MANDELFN CORNERS=4.68/4.76/-.03/.03 FUNCTION=COS" for a graphic demonstration that we're not taking Mandelbrot's name in vain here. We didn't even know these little buggers were here until Mark Peterson found this a few hours before the version incorporating Mandelfns was released.

Note: If you created images using the lambda or mandel "fn" types prior to version 14, and you wish to update the fractal information in the "*.fra" file, simply read the files and save again. You can do this in batch mode via a command line such as:

"fractint oldfile.fra savename=newfile.gif batch=yes"

For example, this procedure can convert a version 13 "type=lambdasine" image to a version 14 "type=lambdafn function=sin" GIF89a image. We do not promise to keep this "backward compatibility" past version 14 - if you want to keep the fractal information in your *.fra files accurate, we recommend conversion. See GIF Save File Format.

1.19 Barnsley Mandelbrot/Julia Sets

(type=barnsleym1/.../j3)

Michael Barnsley has written a fascinating college-level text, "Fractals Everywhere," on fractal geometry and its graphic applications. (See Bibliography.) In it, he applies the principle of the M and J sets to more general functions of two complex variables.

We have incorporated three of Barnsley's examples in Fractint. Their appearance suggests polarized-light microphotographs of minerals, with patterns that are less organic and more crystalline than those of the M/J sets. Each example has both a "Mandelbrot" and a "Julia" type. Toggle between them using the spacebar.

The parameters have the same meaning as they do for the "regular" Mandelbrot and Julia. For types M1, M2, and M3, they are used to "warp"

the image by setting the initial value of Z. For the types J1 through J3, they are the values of C in the generating formulas.

Be sure to try the <O>rbit function while plotting these types.

1.20 Barnsley IFS Fractals

(type=ifs)

One of the most remarkable spin-offs of fractal geometry is the ability to "encode" realistic images in very small sets of numbers -- parameters for a set of functions that map a region of two-dimensional space onto itself. In principle (and increasingly in practice), a scene of any level of complexity and detail can be stored as a handful of numbers, achieving amazing "compression" ratios... how about a super-VGA image of a forest, more than 300,000 pixels at eight bits apiece, from a 1-KB "seed" file?

Again, Michael Barnsley and his co-workers at the Georgia Institute of Technology are to be thanked for pushing the development of these iterated function systems (IFS).

When you select this fractal type, Fractint scans the current IFS file (default is FRACTINT.IFS, a set of definitions supplied with Fractint) for IFS definitions, then prompts you for the IFS name you wish to run. Fern and 3dfern are good ones to start with. You can press <F6> at the selection screen if you want to select a different .IFS file you've written.

Note that some Barnsley IFS values generate images quite a bit smaller than the initial (default) screen. Just bring up the zoom box, center it on the small image, and hit <Enter> to get a full-screen image.

To change the number of dots Fractint generates for an IFS image before stopping, you can change the "maximum iterations" parameter on the <X> options screen.

Fractint supports two types of IFS images: 2D and 3D. In order to fully appreciate 3D IFS images, since your monitor is presumably 2D, we have added rotation, translation, and perspective capabilities. These share values with the same variables used in Fractint's other 3D facilities; for their meaning see Rectangular Coordinate Transformation.
You can enter these values from the command line using:

```
rotation=xrot/yrot/zrot       (try 30/30/30)
shift=xshift/yshift       (shifts BEFORE applying perspective!)
perspective=viewerposition    (try 200)
```

Alternatively, entering <I> from main screen will allow you to modify these values. The defaults are the same as for regular 3D, and are not always optimum for 3D IFS. With the 3dfern IFS type, try rotation=30/30/30. Note that applying shift when using perspective changes the picture -- your "point of view" is moved.

A truly wild variation of 3D may be seen by entering "2" for the stereo mode (see Stereo 3D Viewing),

putting on red/blue "funny glasses", and watching the fern develop
with full depth perception right there before your eyes!

This feature USED to be dedicated to Bruce Goren, as a bribe to get him to
send us MORE knockout stereo slides of 3D ferns, now that we have made it
so easy! Bruce, what have you done for us *LATELY* ?? (Just kidding,
really!)

Each line in an IFS definition (look at FRACTINT.IFS with your editor for
examples) contains the parameters for one of the generating functions,
e.g. in FERN:

```
   a  b     c    d  e    f    p
 ____________________________________
   0   0    0  .16  0    0   .01
 .85   .04 -.04  .85  0  1.6   .85
 .2   -.26  .23  .22  0  1.6   .07
-.15   .28  .26  .24  0  .44   .07
```

The values on each line define a matrix, vector, and probability:

```
    matrix   vector  prob
    |a b|     |e|     p
    |c d|     |f|
```

The "p" values are the probabilities assigned to each function (how often
it is used), which add up to one. Fractint supports up to 32 functions,
although usually three or four are enough.

3D IFS definitions are a bit different. The name is followed by (3D) in
the definition file, and each line of the definition contains 13 numbers:
a b c d e f g h i j k l p, defining:

```
    matrix   vector  prob
    |a b c|   |j|     p
    |d e f|   |k|
    |g h i|   |l|
```

;You can experiment with changes to IFS definitions interactively by using
;Fractint's <Z> command. After selecting an IFS definition, hit <Z> to
;bring up the IFS editor. This editor displays the current IFS values, lets
;you modify them, and lets you save your modified values as a text file
;which you can then merge into an XXX.IFS file for future use with
;Fractint.
;
The program FDESIGN can be used to design IFS fractals - see
FDESIGN.

You can save the points in your IFS fractal in the file ORBITS.RAW which is
overwritten each time a fractal is generated. The program Acrospin can
read this file and will let you view the fractal from any angle using
the cursor keys. See Acrospin.

1.21 Sierpinski Gasket

(type=sierpinski)

Another pre-Mandelbrot classic, this one found by W. Sierpinski around
World War I. It is generated by dividing a triangle into four congruent
smaller triangles, doing the same to each of them, and so on, yea, even
unto infinity. (Notice how hard we try to avoid reiterating "iterating"?)

If you think of the interior triangles as "holes", they occupy more and
more of the total area, while the "solid" portion becomes as hopelessly
fragile as that gasket you HAD to remove without damaging it -- you
remember, that Sunday afternoon when all the parts stores were closed?
There's a three-dimensional equivalent using nested tetrahedrons instead
of triangles, but it generates too much pyramid power to be safely
unleashed yet.

There are no parameters for this type. We were able to implement it with
integer math routines, so it runs fairly quickly even without an FPU.

1.22 Quartic Mandelbrot/Julia

(type=mandel4/julia4)

These fractal types are the moral equivalent of the original M and J sets,
except that they use the formula Z(n+1) = Z(n)^4 + C, which adds
additional pseudo-symmetries to the plots. The "Mandel4" set maps to the
"Julia4" set via -- surprise! -- the spacebar toggle. The M4 set is kind
of boring at first (the area between the "inside" and the "outside" of the
set is pretty thin, and it tends to take a few zooms to get to any
interesting sections), but it looks nice once you get there. The Julia
sets look nice right from the start.

Other powers, like Z(n)^3 or Z(n)^7, work in exactly the same fashion. We
used this one only because we're lazy, and Z(n)^4 = (Z(n)^2)^2.

1.23 Distance Estimator

(distest=nnn/nnn)

This used to be type=demm and type=demj. These types have not died, but
are only hiding! They are equivalent to the mandel and julia types with
the "distest=" option selected with a predetermined value.

The Distance Estimator Method
can be used to produce higher quality images of M and J sets,
especially suitable for printing in black and white.

If you have some *.fra files made with the old types demm/demj, you may
want to convert them to the new form. See the Mandelfn
section for directions to carry out the conversion.

1.24 Pickover Mandelbrot/Julia Types

(type=manfn+zsqrd/julfn+zsqrd, manzpowr/julzpowr, manzzpwr/julzzpwr, manfn+exp/julfn+exp - formerly included man/julsinzsqrd and man/julsinexp which have now been generalized)

These types have been explored by Clifford A. Pickover, of the IBM Thomas J. Watson Research center. As implemented in Fractint, they are regular Mandelbrot/Julia set pairs that may be plotted with or without the biomorph option Pickover used to create organic-looking beasties (see below).
These types are produced with formulas built from the functions z^z, z^n, sin(z), and e^z for complex z. Types with "power" or "pwr" in their name have an exponent value as a third parameter. For example, type=manzpower params=0/0/2 is our old friend the classical Mandelbrot, and type=manzpower params=0/0/4 is the Quartic Mandelbrot. Other values of the exponent give still other fractals. Since these WERE the original "biomorph" types, we should give an example. Try:

```
FRACTINT type=manfn+zsqrd biomorph=0 corners=-8/8/-6/6 function=sin
```

to see a big biomorph digesting little biomorphs!

1.25 Pickover Popcorn

(type=popcorn/popcornjul)

Here is another Pickover idea. This one computes and plots the orbits of the dynamic system defined by:

```
x(n+1) = x(n) - h*sin(y(n)+tan(3*y(n))
y(n+1) = y(n) - h*sin(x(n)+tan(3*x(n))
```

with the initializers x(0) and y(0) equal to ALL the complex values within the "corners" values, and h=.01. ALL these orbits are superimposed, resulting in "popcorn" effect. You may want to use a maxiter value less than normal - Pickover recommends a value of 50. As a bonus, type=popcornjul shows the Julia set generated by these same equations with the usual escape-time coloring. Turn on orbit viewing with the "O" command, and as you watch the orbit pattern you may get some insight as to where the popcorn comes from. Although you can zoom and rotate popcorn, the results may not be what you'd expect, due to the superimposing of orbits and arbitrary use of color. Just for fun we added type popcornjul, which is the plain old Julia set calculated from the same formula.

1.26 Dynamic System

(type=dynamic, dynamic2)

These fractals are based on a cyclic system of differential equations:

```
x'(t) = -f(y(t))
y'(t) = f(x(t))
```

These equations are approximated by using a small time step dt, forming

```
a time-discrete dynamic system:
   x(n+1) = x(n) - dt*f(y(n))
   y(n+1) = y(n) + dt*f(x(n))
The initial values x(0) and y(0) are set to various points in the plane,
the dynamic system is iterated, and the resulting orbit points are plotted.

In fractint, the function f is restricted to:
      f(k) = sin(k + a*fn1(b*k))
The parameters are the spacing of the initial points, the time step dt,
and the parameters (a,b,fn1) that affect the function f.
Normally the orbit points are plotted individually, but for a negative
spacing the points are connected.

This fractal is similar to the
                Pickover Popcorn
                .

A variant is the implicit Euler approximation:
   y(n+1) = y(n) + dt*f(x(n))
   x(n+1) = x(n) - dt*f(y(n+1))
This variant results in complex orbits.  The implicit Euler approximation
is selected by entering dt<0.

There are two options that have unusual effects on these fractals.  The
Orbit Delay value controls how many initial points are computed before
the orbits are displayed on the screen.  This allows the orbit to settle
down.  The outside=summ option causes each pixel to increment color every
time an orbit touches it; the resulting display is a 2-d histogram.

These fractals are discussed in Chapter 14 of Pickover's "Computers,
Pattern, Chaos, and Beauty".
```

1.27 Mandelcloud

```
(type=mandelcloud)

This fractal computes the Mandelbrot function, but displays it differently.
It starts with regularly spaced initial pixels and displays the resulting
orbits.  This idea is somewhat similar to the Dynamic System.

There are two options that have unusual effects on this fractal.  The
Orbit Delay value controls how many initial points are computed before
the orbits are displayed on the screen.  This allows the orbit to settle
down.  The outside=summ option causes each pixel to increment color every
time an orbit touches it; the resulting display is a 2-d histogram.

This fractal was invented by Noel Giffin.
```

1.28 Peterson Variations

```
(type=marksmandel, marksjulia, cmplxmarksmand, cmplxmarksjul, marksmandelpwr,
tim's_error)
```

These fractal types are contributions of Mark Peterson. MarksMandel and MarksJulia are two families of fractal types that are linked in the same manner as the classic Mandelbrot/Julia sets: each MarksMandel set can be considered as a mapping into the MarksJulia sets, and is linked with the spacebar toggle. The basic equation for these sets is:

```
      Z(n+1) = ((lambda^exp) * Z(n)^2) + lambda
```

where Z(0) = 0.0 and lambda is (x + iy) for MarksMandel. For MarksJulia, Z(0) = (x + iy) and lambda is a constant (taken from the MarksMandel spacebar toggle, if that method is used). The exponent is a positive integer or a complex number. We call these "families" because each value of the exponent yields a different MarksMandel set, which turns out to be a kinda-polygon with (exponent+1) sides. The exponent value is the third parameter, after the "initialization warping" values. Typically one would use null warping values, and specify the exponent with something like "PARAMS=0/0/4", which creates an unwarped, pentagonal MarksMandel set.

In the process of coding MarksMandelPwr formula type, Tim Wegner created the type "tim's_error" after making an interesting coding mistake.

1.29 Unity

```
                (type=unity)
```

This Peterson variation began with curiosity about other "Newton-style" approximation processes. A simple one,

```
   One = (x * x) + (y * y); y = (2 - One) * x;   x = (2 - One) * y;
```

produces the fractal called Unity.

One of its interesting features is the "ghost lines." The iteration loop bails out when it reaches the number 1 to within the resolution of a screen pixel. When you zoom a section of the image, the bailout criterion is adjusted, causing some lines to become thinner and others thicker.

Only one line in Unity that forms a perfect circle: the one at a radius of 1 from the origin. This line is actually infinitely thin. Zooming on it reveals only a thinner line, up (down?) to the limit of accuracy for the algorithm. The same thing happens with other lines in the fractal, such as those around |x| = |y| = (1/2)^(1/2) = .7071

Try some other tortuous approximations using the
```
               TEST stub
                and
```
let us know what you come up with!

1.30 Scott Taylor / Lee Skinner Variations

(type=fn(z*z), fn*fn, fn*z+z, fn+fn, sqr(1/fn), sqr(fn), spider, tetrate, manowar)

```
Two of Fractint's faithful users went bonkers when we introduced the
"formula" type, and came up with all kinds of variations on escape-time
fractals using trig functions.  We decided to put them in as regular
types, but there were just too many! So we defined the types with variable
functions and let you, the, overwhelmed user, specify what the functions
should be! Thus Scott Taylor's "z = sin(z) + z^2" formula type is now the
"fn+fn" regular type, and EITHER function can be one of sin, cos, tan, cotan,
sinh, cosh, tanh, cotanh, exp, log, sqr, recip, ident, conj, flip, or cosxx.
Plus we give you 4 parameters to set, the complex
coefficients of the two functions!  Thus the innocent-looking "fn+fn" type
is really 256 different types in disguise, not counting the damage
done by the parameters!

 Some functions that require further explanation:

 conj()   - returns the complex conjugate of the argument. That is, changes
            sign of the imaginary component of argument: (x,y) becomes (x,-y)
 ident()  - identity function. Leaves the value of the argument unchanged,
            acting like a "z" term in a formula.

 flip()   - Swap the real and imaginary components of the complex number.
            e.g. (4,5) would become (5,4)

Lee informs us that you should not judge fractals by their "outer"
appearance. For example, the images produced by z = sin(z) + z^2 and z =
sin(z) - z^2 look very similar, but are different when you zoom in.
```

1.31 Kam Torus

```
(type=kamtorus, kamtorus3d)

This type is created by superimposing orbits generated by a set of
equations, with a variable incremented each time.

   x(0) = y(0) = orbit/3;
   x(n+1) = x(n)*cos(a) + (x(n)*x(n)-y(n))*sin(a)
   y(n+1) = x(n)*sin(a) - (x(n)*x(n)-y(n))*cos(a)

After each orbit, 'orbit' is incremented by a step size. The parameters
are angle "a", step size for incrementing 'orbit', stop value for 'orbit',
and points per orbit. Try this with a stop value of 5 with sound=x for
some weird fractal music (ok, ok, fractal noise)! You will also see the
KAM Torus head into some chaotic territory that Scott Taylor wanted to
hide from you by setting the defaults the way he did, but now we have
revealed all!

The 3D variant is created by treating 'orbit' as the z coordinate.

With both variants, you can adjust the "maxiter" value (<X> options
screen or parameter maxiter=) to change the number of orbits plotted.
```

1.32 Bifurcation

(type=bifxxx)

The wonder of fractal geometry is that such complex forms can arise from such simple generating processes. A parallel surprise has emerged in the study of dynamical systems: that simple, deterministic equations can yield chaotic behavior, in which the system never settles down to a steady state or even a periodic loop. Often such systems behave normally up to a certain level of some controlling parameter, then go through a transition in which there are two possible solutions, then four, and finally a chaotic array of possibilities.

This emerged many years ago in biological models of population growth. Consider a (highly over-simplified) model in which the rate of growth is partly a function of the size of the current population:

New Population = Growth Rate * Old Population * (1 - Old Population)

where population is normalized to be between 0 and 1. At growth rates less than 200 percent, this model is stable: for any starting value, after several generations the population settles down to a stable level. But for rates over 200 percent, the equation's curve splits or "bifurcates" into two discrete solutions, then four, and soon becomes chaotic.

Type=bifurcation illustrates this model. (Although it's now considered a poor one for real populations, it helped get people thinking about chaotic systems.) The horizontal axis represents growth rates, from 190 percent (far left) to 400 percent; the vertical axis normalized population values, from 0 to 4/3. Notice that within the chaotic region, there are narrow bands where there is a small, odd number of stable values. It turns out that the geometry of this branching is fractal; zoom in where changing pixel colors look suspicious, and see for yourself.

Three parameters apply to bifurcations: Filter Cycles, Seed Population, and Function or Beta.

Filter Cycles (default 1000) is the number of iterations to be done before plotting maxiter population values. This gives the iteration time to settle into the characteristic patterns that constitute the bifurcation diagram, and results in a clean-looking plot. However, using lower values produces interesting results too. Set Filter Cycles to 1 for an unfiltered map.

Seed Population (default 0.66) is the initial population value from which all others are calculated. For filtered maps the final image is independent of Seed Population value in the valid range (0.0 < Seed Population < 1.0).

Seed Population becomes effective in unfiltered maps - try setting Filter Cycles to 1 (unfiltered) and Seed Population to 0.001 ("PARAMS=1/.001" on the command line). This results in a map overlaid with nice curves. Each Seed Population value results in a different set of curves.

Function (default "ident") is the function applied to the old population before the new population is determined. The "ident" function calculates the same bifurcation fractal that was generated before these formulae were generalized.

Beta is used in the bifmay bifurcations and is the power to which the

denominator is raised.

Note that fractint normally uses periodicity checking to speed up
bifurcation computation. However, in some cases a better quality image
will be obtained if you turn off periodicity checking with "periodicity=no";
for instance, if you use a high number of iterations and a smooth
colormap.

Many formulae can be used to produce bifurcations. Mitchel Feigenbaum
studied lots of bifurcations in the mid-70's, using a HP-65 calculator
(IBM PCs, Fractals, and Fractint, were all Sci-Fi then !). He studied
where bifurcations occurred, for the formula r*p*(1-p), the one described
above. He found that the ratios of lengths of adjacent areas of
bifurcation were four and a bit. These ratios vary, but, as the growth
rate increases, they tend to a limit of 4.669+. This helped him guess
where bifurcation points would be, and saved lots of time.

When he studied bifurcations of r*sin(PI*p) he found a similar pattern,
which is not surprising in itself. However, 4.669+ popped out, again.
Different formulae, same number ? Now, THAT's surprising ! He tried many
other formulae and ALWAYS got 4.669+ - Hot Damn !!! So hot, in fact, that
he phoned home and told his Mom it would make him Famous ! He also went on
to tell other scientists. The rest is History...

(It has been conjectured that if Feigenbaum had a copy of Fractint, and
used it to study bifurcations, he may never have found his Number, as it
only became obvious from long perusal of hand-written lists of values,
without the distraction of wild color-cycling effects !).

We now know that this number is as universal as PI or E. It appears in
situations ranging from fluid-flow turbulence, electronic oscillators,
chemical reactions, and even the Mandelbrot Set - yup, fraid so:
"budding" of the Mandelbrot Set along the negative real axis occurs at
intervals determined by Feigenbaum's Number, 4.669201660910.....

Fractint does not make direct use of the Feigenbaum Number (YET !).
However, it does now reflect the fact that there is a whole sub-species of
Bifurcation-type fractals. Those implemented to date, and the related
formulae, (writing P for pop[n+1] and p for pop[n]) are :

```
  bifurcation  P =  p + r*fn(p)*(1-fn(p))  Verhulst Bifurcations.
  biflambda    P =      r*fn(p)*(1-fn(p))  Real equivalent of Lambda Sets.
  bif+sinpi    P =  p + r*fn(PI*p)         Population scenario based on...
  bif=sinpi    P =      r*fn(PI*p)         ...Feigenbaum's second formula.
  bifstewart   P =      r*fn(p)*fn(p) - 1  Stewart Map.
  bifmay       P =      r*p / ((1+p)^b)    May Map.
```

It took a while for bifurcations to appear here, despite them being over a
century old, and intimately related to chaotic systems. However, they are
now truly alive and well in Fractint!

1.33 Orbit Fractals

Orbit Fractals are generated by plotting an orbit path in two or ↩
three

dimensional space.

See
Lorenz Attractors
,
Rossler Attractors
,

Henon Attractors
,
Pickover Attractors
,
Gingerbreadman
,
and
Martin Attractors
.

The orbit trajectory for these types can be saved in the file ORBITS.RAW by invoking
Fractint with the "orbitsave=yes" command-line option. This file will be overwritten each time you generate a new fractal, so rename it if you want to save it. A nifty program called Acrospin can read these files and rapidly rotate them in 3-D - see Acrospin.

1.34 Lorenz Attractors

(type=lorenz/lorenz3d)

The "Lorenz Attractor" is a "simple" set of three deterministic equations developed by Edward Lorenz while studying the non- repeatability of weather patterns. The weather forecaster's basic problem is that even very tiny changes in initial patterns ("the beating of a butterfly's wings" - the official term is "sensitive dependence on initial conditions") eventually reduces the best weather forecast to rubble.

The lorenz attractor is the plot of the orbit of a dynamic system consisting of three first order non-linear differential equations. The solution to the differential equation is vector-valued function of one variable. If you think of the variable as time, the solution traces an orbit. The orbit is made up of two spirals at an angle to each other in three dimensions. We change the orbit color as time goes on to add a little dazzle to the image. The equations are:

```
dx/dt = -a*x + a*y
dy/dt =  b*x - y   -z*x
dz/dt = -c*z + x*y
```

We solve these differential equations approximately using a method known as the first order taylor series. Calculus teachers everywhere will kill us for saying this, but you treat the notation for the derivative dx/dt as though it really is a fraction, with "dx" the small change in x that happens when the time changes "dt". So multiply through the above equations by dt, and you will have the change in the orbit for a small time step. We add these changes to the old vector to get the new vector

after one step. This gives us:

```
xnew = x + (-a*x*dt) + (a*y*dt)
ynew = y + (b*x*dt) - (y*dt) - (z*x*dt)
znew = z + (-c*z*dt) + (x*y*dt)

(default values: dt = .02, a = 5, b = 15, c = 1)
```

We connect the successive points with a line, project the resulting 3D orbit onto the screen, and voila! The Lorenz Attractor!

We have added two versions of the Lorenz Attractor. "Type=lorenz" is the Lorenz attractor as seen in everyday 2D. "Type=lorenz3d" is the same set of equations with the added twist that the results are run through our perspective 3D routines, so that you get to view it from different angles (you can modify your perspective "on the fly" by using the <I> command.) If you set the "stereo" option to "2", and have red/blue funny glasses on, you will see the attractor orbit with depth perception.

Hint: the default perspective values (x = 60, y = 30, z = 0) aren't the best ones to use for fun Lorenz Attractor viewing. Experiment a bit - start with rotation values of 0/0/0 and then change to 20/0/0 and 40/0/0 to see the attractor from different angles.- and while you're at it, use a non-zero perspective point Try 100 and see what happens when you get *inside* the Lorenz orbits. Here comes one - Duck! While you are at it, turn on the sound with the "X". This way you'll at least hear it coming!

Different Lorenz attractors can be created using different parameters. Four parameters are used. The first is the time-step (dt). The default value is .02. A smaller value makes the plotting go slower; a larger value is faster but rougher. A line is drawn to connect successive orbit values. The 2nd, third, and fourth parameters are coefficients used in the differential equation (a, b, and c). The default values are 5, 15, and 1. Try changing these a little at a time to see the result.

1.35 Rossler Attractors

(type=rossler3D)

This fractal is named after the German Otto Rossler, a non-practicing medical doctor who approached chaos with a bemusedly philosophical attitude. He would see strange attractors as philosophical objects. His fractal namesake looks like a band of ribbon with a fold in it. All we can say is we used the same calculus-teacher-defeating trick of multiplying the equations by "dt" to solve the differential equation and generate the orbit. This time we will skip straight to the orbit generator - if you followed what we did above with type Lorenz you can easily reverse engineer the differential equations.

```
xnew = x - y*dt -   z*dt
ynew = y + x*dt + a*y*dt
znew = z + b*dt + x*z*dt - c*z*dt
```

Default parameters are dt = .04, a = .2, b = .2, c = 5.7

1.36 Henon Attractors

(type=henon)

Michel Henon was an astronomer at Nice observatory in southern France. He came to the subject of fractals via investigations of the orbits of astronomical objects. The strange attractor most often linked with Henon's name comes not from a differential equation, but from the world of discrete mathematics - difference equations. The Henon map is an example of a very simple dynamic system that exhibits strange behavior. The orbit traces out a characteristic banana shape, but on close inspection, the shape is made up of thicker and thinner parts. Upon magnification, the thicker bands resolve to still other thick and thin components. And so it goes forever! The equations that generate this strange pattern perform the mathematical equivalent of repeated stretching and folding, over and over again.

```
xnew =  1 + y - a*x*x
ynew =  b*x
```

The default parameters are a=1.4 and b=.3.

1.37 Pickover Attractors

(type=pickover)

Clifford A. Pickover of the IBM Thomas J. Watson Research center is such a creative source for fractals that we attach his name to this one only with great trepidation. Probably tomorrow he'll come up with another one and we'll be back to square one trying to figure out a name!

This one is the three dimensional orbit defined by:

```
xnew = sin(a*y) - z*cos(b*x)
ynew = z*sin(c*x) - cos(d*y)
znew = sin(x)
```

Default parameters are: a = 2.24, b = .43, c = -.65, d = -2.43

1.38 Gingerbreadman

(type=gingerbreadman)

This simple fractal is a charming example stolen from "Science of Fractal Images", p. 149.

```
xnew = 1 - y + |x|
ynew = x
```

The initial x and y values are set by parameters, defaults x=-.1, y = 0.

1.39 Martin Attractors

```
(type=hopalong/martin)

These fractal types are from A. K. Dewdney's "Computer Recreations" column
in "Scientific American". They are attributed to Barry Martin of Aston
University in Birmingham, England.

Hopalong is an "orbit" type fractal like lorenz. The image is obtained by
iterating this formula after setting z(0) = y(0) = 0:
      x(n+1) = y(n) - sign(x(n))*sqrt(abs(b*x(n)-c))
      y(n+1) = a - x(n)
Parameters are a, b, and c. The function "sign()"  returns 1 if the argument
is positive, -1 if argument is negative.

This fractal continues to develop in surprising ways after many iterations.

Another Martin fractal is simpler. The iterated formula is:
      x(n+1) = y(n) - sin(x(n))
      y(n+1) = a - x(n)
The parameter is "a". Try values near the number pi.
```

1.40 Icon

```
(type=icon/icon3d)

  This fractal type was inspired by the book "Symmetry in Chaos"
  by Michael Field and Martin Golubitsky (ISBN 0-19-853689-5, Oxford Press)

  To quote from the book's jacket,

    "Field and Golubitsky describe how a chaotic process eventually can
    lead to symmetric patterns (in a river, for instance, photographs of
    the turbulent movement of eddies, taken over time, often reveal
    patterns on the average."

  The Icon type implemented here maps the classic population logistic
  map of bifurcation fractals onto the complex plane in Dn symmetry.

  The initial points plotted are the more chaotic initial orbits, but
  as you wait, delicate webs will begin to form as the orbits settle
  into a more periodic pattern.  Since pixels are colored by the number
  of times they are hit, the more periodic paths will become clarified
  with time.  These fractals run continuously.

There are 6 parameters:  Lambda, Alpha, Beta, Gamma, Omega, and Degree
    Omega  0 = Dn, or dihedral (rotation + reflectional) symmetry
          !0 = Zn, or cyclic (rotational) symmetry
    Degree = n, or Degree of symmetry
```

1.41 Quaternion

(type=quat,quatjul)

These fractals are based on quaternions. Quaternions are an extension of complex numbers, with 4 parts instead of 2. That is, a quaternion Q equals a+ib+jc+kd, where a,b,c,d are reals. Quaternions have rules for addition and multiplication. The normal Mandelbrot and Julia formulas can be generalized to use quaternions instead of complex numbers.

There is one complication. Complex numbers have 2 parts, so they can be displayed on a plane. Quaternions have 4 parts, so they require 4 dimensions to view. That is, the quaternion Mandelbrot set is actually a 4-dimensional object. Each quaternion C generates a 4-dimensional Julia set.

One method of displaying the 4-dimensional object is to take a 3-dimensional slice and render the resulting object in 3-dimensional perspective. Fractint isn't that sophisticated, so it merely displays a 2-dimensional slice of the resulting object. (Note: Now Fractint is that sophisticated! See the Julibrot type!)

In fractint, for the Julia set, you can specify the four parameters of the quaternion constant: c=(c1,ci,cj,ck), but the 2-dimensional slice of the z-plane Julia set is fixed to (xpixel,ypixel,0,0).

For the Mandelbrot set, you can specify the position of the c-plane slice: (xpixel,ypixel,cj,ck).

These fractals are discussed in Chapter 10 of Pickover's "Computers, Pattern, Chaos, and Beauty".

1.42 HyperComplex

(type=hypercomplex,hypercomplexj)

These fractals are based on hypercomplex numbers, which like quaternions are a four dimensional generalization of complex numbers. It is not possible to fully generalize the complex numbers to four dimensions without sacrificing some of the algebraic properties shared by real and complex numbers. Quaternions violate the commutative law of multiplication, which says z1*z2 = z2*z1. Hypercomplex numbers fail the rule that says all non-zero elements have multiplicative inverses - that is, if z is not 0, there should be a number 1/z such that (1/z)*(z) = 1. This law holds most of the time but not all the time for hypercomplex numbers.

However hypercomplex numbers have a wonderful property for fractal purposes. Every function defined for complex numbers has a simple generalization to hypercomplex numbers. Fractint's implementation takes advantage of this by using "fn" variables - the iteration formula is

    h(n+1) = fn(h(n)) + C.

where "fn" is the hypercomplex generalization of sin, cos, log, sqr etc.

You can see 3D versions of these fractals using fractal type Julibrot. Hypercomplex numbers were brought to our attention by Clyde Davenport, author of "A Hypercomplex Calculus with Applications to Relativity", ISBN 0-9623837-0-8.

1.43 Cellular Automata

(type=cellular)

These fractals are generated by 1-dimensional cellular automata. Consider a 1-dimensional line of cells, where each cell can have the value 0 or 1. In each time step, the new value of a cell is computed from the old value of the cell and the values of its neighbors. On the screen, each horizontal row shows the value of the cells at any one time. The time axis proceeds down the screen, with each row computed from the row above.

Different classes of cellular automata can be described by how many different states a cell can have (k), and how many neighbors on each side are examined (r). Fractint implements the binary nearest neighbor cellular automata (k=2,r=1), the binary next-nearest neighbor cellular automata (k=2,r=2), and the ternary nearest neighbor cellular automata (k=3,r=1) and several others.

The rules used here determine the next state of a given cell by using the sum of the states in the cell's neighborhood. The sum of the cells in the neighborhood are mapped by rule to the new value of the cell. For the binary nearest neighbor cellular automata, only the closest neighbor on each side is used. This results in a 4 digit rule controlling the generation of each new line: if each of the cells in the neighborhood is 1, the maximum sum is 1+1+1 = 3 and the sum can range from 0 to 3, or 4 values. This results in a 4 digit rule. For instance, in the rule 1010, starting from the right we have 0->0, 1->1, 2->0, 3->1. If the cell's neighborhood sums to 2, the new cell value would be 0.

For the next-nearest cellular automata (kr = 22), each pixel is determined from the pixel value and the two neighbors on each side. This results in a 6 digit rule.

For the ternary nearest neighbor cellular automata (kr = 31), each cell can have the value 0, 1, or 2. A single neighbor on each side is examined, resulting in a 7 digit rule.

kr	#'s in rule	example rule	kr	#'s in rule	example rule
21	4	1010	42	16	2300331230331001
31	7	1211001	23	8	10011001
41	10	3311100320	33	15	021110101210010
51	13	2114220444030	24	10	0101001110
61	16	3452355321541340	25	12	110101011001
22	6	011010	26	14	00001100000110
32	11	21212002010	27	16	0010000000000110

The starting row of cells can be set to a pattern of up to 16 digits or to a random pattern. The borders are set to zeros if a pattern is entered or are set randomly if the starting row is set randomly.

A zero rule will randomly generate the rule to use.

Hitting the space bar toggles between continuously generating the cellular automata and stopping at the end of the current screen.

Recommended reading:

"Computer Software in Science and Mathematics", Stephen Wolfram, Scientific American, September, 1984.
"Abstract Mathematical Art", Kenneth E. Perry, BYTE, December, 1986.
"The Armchair Universe", A. K. Dewdney, W. H. Freeman and Company, 1988.
"Complex Patterns Generated by Next Nearest Neighbors Cellular Automata", Wentian Li, Computers & Graphics, Volume 13, Number 4.

1.44 test

(type=test)

This is a stub that we (and you!) use for trying out new fractal types. "Type=test" fractals make use of Fractint's structure and features for whatever code is in the routine 'testpt()' (located in the small source file TESTPT.C) to determine the color of a particular pixel.

If you have a favorite fractal type that you believe would fit nicely into Fractint, just rewrite the C function in TESTPT.C (or use the prototype function there, which is a simple M-set implementation) with an algorithm that computes a color based on a point in the complex plane.

After you get it working, send your code to one of the authors and we might just add it to the next release of Fractint, with full credit to you. Our criteria are: 1) an interesting image and 2) a formula significantly different from types already supported. (Bribery may also work. THIS author is completely honest, but I don't trust those other guys.) Be sure to include an explanation of your algorithm and the parameters supported, preferably formatted as you see here to simplify folding it into the documentation.

1.45 Formula

(type=formula)

This is a "roll-your-own" fractal interpreter - you don't even need a compiler!

To run a "type=formula" fractal, you first need a text file containing formulas (there's a sample file - FRACTINT.FRM - included with this distribution). When you select the "formula" fractal type, Fractint scans the current formula file (default is FRACTINT.FRM) for formulas, then prompts you for the formula name you wish to run. After prompting for any parameters, the formula is parsed for syntax errors and then the fractal is generated. If you want to use a different formula file, press <F6> when you are prompted to select a formula name.

There are two command-line options that work with type=formula ("formulafile=" and "formulaname="), useful when you are using this fractal type in batch mode.

The following documentation is supplied by Mark Peterson, who wrote the formula interpreter:

Formula fractals allow you to create your own fractal formulas. The general format is:

```
   Mandelbrot(XAXIS) { z = Pixel:  z = sqr(z) + pixel, |z| <= 4 }
      |   |    |        |      |
     Name     Symmetry    Initial   Iteration   Bailout
        Condition       Criteria
```

Initial conditions are set, then the iterations performed until the bailout criteria is true or 'z' turns into a periodic loop.
All variables are created automatically by their usage and treated as complex. If you declare 'v = 2' then the variable 'v' is treated as a complex with an imaginary value of zero.

```
    Predefined Variables (x, y)
    -------------------------------------------
    z    used for periodicity checking
    p1     parameters 1 and 2
    p2     parameters 3 and 4
    pixel    screen coordinates
          LastSqr        Modulus from the last sqr() function
          rand           Complex random number

          Precedence
          -------------------------------------------
          1              sin(), cos(), sinh(), cosh(), cosxx(),
                         tan(), cotan(), tanh(), cotanh(),
                         sqr, log(), exp(), abs(), conj(), real(),
                         imag(), flip(), fn1(), fn2(), fn3(), fn4(),
                         srand()
          2              - (negation), ^ (power)
          3              * (multiplication), / (division)
          4              + (addition), - (subtraction)
          5              = (assignment)
          6              < (less than), <= (less than or equal to)
                         > (greater than), >= (greater than or equal to)
                         == (equal to), != (not equal to)
          7              && (logical AND), || (logical OR)
```

Precedence may be overridden by use of parenthesis. Note the modulus squared operator |z| is also parenthetic and always sets the imaginary component to zero. This means 'c * |z - 4|' first subtracts 4 from z, calculates the modulus squared then multiplies times 'c'. Nested modulus squared operators require overriding parenthesis:

```
    c * |z + (|pixel|)|
```

The functions fn1(...) to fn4(...) are variable functions - when used, the user is prompted at run time (on the <Z> screen) to specify one of sin, cos, sinh, cosh, exp, log, sqr, etc. for each required variable function.

The formulas are performed using either integer or floating point mathematics depending on the <F> floating point toggle. If you do not have an FPU then type MPC math is performed in lieu of traditional floating point.

The 'rand' predefined variable is changed with each iteration to a new

random number with the real and imaginary components containing a value between zero and 1. Use the srand() function to initialize the random numbers to a consistent random number sequence. If a formula does not contain the srand() function, then the formula compiler will use the system time to initialize the sequence. This could cause a different fractal to be generated each time the formula is used depending on how the formula is written.

Remember that when using integer math there is a limited dynamic range, so what you think may be a fractal could really be just a limitation of the integer math range. God may work with integers, but His dynamic range is many orders of magnitude greater than our puny 32 bit mathematics! Always verify with the floating point <F> toggle.

1.46 Frothy Basins

(type=frothybasin)

Frothy Basins, or Riddled Basins, were discovered by James C. Alexander of the University of Maryland. The discussion below is derived from a two page article entitled "Basins of Froth" in Science News, November 14, 1992 and from correspondence with others, including Dr. Alexander.

The equations that generate this fractal are not very different from those that generate many other orbit fractals.

```
z(0) = pixel;
z(n+1) = z(n)^2 - c*conj(z(n))
where c = 1 + ai,  and  a = 1.02871376822...
```

One of the things that makes this fractal so interesting is the shape of the dynamical system's attractors. It is not at all uncommon for a dynamical system to have non-point attractors. Shapes such as circles are very common. Strange attractors are attractors which are themselves fractal. What is unusual about this system, however, is that the attractors intersect. This is the first case in which such a phenomenon has been observed. The three attractors for this system are made up of line segments which overlap to form an equilateral triangle. This attractor triangle can be seen by pressing the 'o' key while the fractal is being generated to turn on the "show orbits" option.

An interesting variation on this fractal can be generated by applying the above mapping twice per each iteration. The result is that each of the three attractors is split into two parts, giving the system six attractors.

These are also called "Riddled Basins" because each basin is riddled with holes. Which attractor a point is eventually pulled into is extremely sensitive to its initial position. A very slight change in any direction may cause it to end up on a different attractor. As a result, the basins are thoroughly intermingled. The effect appears to be a frothy mixture that has been subjected to lots of stirring and folding.

Pixel color is determined by which attractor captures the orbit. The shade of color is determined by the number of iterations required to capture the

orbit. In Fractint, the actual shade of color used depends on how many colors are available in the video mode being used.

If 256 colors are available, the default coloring scheme is determined by the number of iterations that were required to capture the orbit. An alternative coloring scheme can be used where the shade is determined by the iterations required divided by the maximum iterations. This method is especially useful on deeply zoomed images.

If only 16 colors are available, then only the alternative coloring scheme is used. If fewer than 16 colors are available, then Fractint just colors the basins without any shading.

1.47 Julibrots

(type=julibrot)

The Julibrot fractal type uses a general-purpose renderer for visualizing three dimensional solid fractals. Originally Mark Peterson developed this rendering mechanism to view a 3-D sections of a 4-D structure he called a "Julibrot". This structure, also called "layered Julia set" in the fractal literature, hinges on the relationship between the Mandelbrot and Julia sets. Each Julia set is created using a fixed value c in the iterated formula z^2 + c. The Julibrot is created by layering Julia sets in the x-y plane and continuously varying c, creating new Julia sets as z is incremented. The solid shape thus created is rendered by shading the surface using a brightness inversely proportional to the virtual viewer's eye.

Starting with Fractint version 18, the Julibrot engine can be used with other Julia formulas besides the classic z^2 + c. The first field on the Julibrot parameter screen lets you select which orbit formula to use.

You can also use the Julibrot renderer to visualize 3D cross sections of true four dimensional Quaternion and Hypercomplex fractals.

The Julibrot Parameter Screens

Orbit Algorithm - select the orbit algorithm to use. The available possibilities include 2-D Julia and both mandelbrot and Julia variants of the 4-D Quaternion and Hypercomplex fractals.

Orbit parameters - the next screen lets you fill in any parameters belonging to the orbit algorithm. This list of parameters is not necessarily the same as the list normally presented for the orbit algorithm, because some of these parameters are used in the Julibrot layering process.

From/To Parameters
These parameters allow you to specify the "Mandelbrot" values used to generate the layered Julias. The parameter c in the Julia formulas will be incremented in steps ranging from the "from" x and y values to the "to" x and y values. If the orbit formula is one of the "true" four dimensional fractal types quat, quatj, hypercomplex, or hypercomplexj, then these numbers are used with the 3rd and 4th dimensional values.

```
   The "from/to" variables are different for the different kinds of orbit
   algorithm.

      2D Julia sets - complex number formula z' = f(z) + c
         The "from/to" parameters change the values of c.
      4D Julia sets - Quaternion or Hypercomplex formula z' = f(z) + c
         The four dimensions of c are set by the orbit parameters.
         The first two dimensions of z are determined by the corners values.
         The third and fourth dimensions of z are the "to/from" variables.
      4D Mandelbrot sets - Quaternion or Hypercomplex formula z' = f(z) + c
         The first two dimensions of c are determined by the corners values.
         The third and fourth dimensions of c are the "to/from" variables.

Distance between the eyes - set this to 2.5 if you want a red/blue
   anaglyph image, 0 for a normal greyscale image.

Number of z pixels - this sets how many layers are rendered in the screen
   z-axis. Use a higher value with higher resolution video modes.

The remainder of the parameters are needed to construct the red/blue
picture so that the fractal appears with the desired depth and proper 'z'
location.  With the origin set to 8 inches beyond the screen plane and the
depth of the fractal at 8 inches the default fractal will appear to start
at 4 inches beyond the screen and extend to 12 inches if your eyeballs are
2.5 inches apart and located at a distance of 24 inches from the screen.
The screen dimensions provide the reference frame.
```

1.48 Diffusion Limited Aggregation

```
(type=diffusion)

This type begins with a single point in the center of the screen.
Subsequent points move around randomly until coming into contact with the
first point, at which time their locations are fixed and they are colored
randomly.  This process repeats until the fractals reaches the edge of the
screen.  Use the show orbits function to see the points' random motion.

One unfortunate problem is that on a large screen, this process will tend
to take eons.  To speed things up, the points are restricted to a box
around the initial point.  The first and only parameter to diffusion
contains the size of the border between the fractal and the edge of the
box.  If you make this number small, the fractal will look more solid and
will be generated more quickly.

Diffusion was inspired by a Scientific American article a couple of years
back which includes actual pictures of real physical phenomena that behave
like this.

Thanks to Adrian Mariano for providing the diffusion code and
documentation. Juan J. Buhler added the additional options.
```

1.49 Lyapunov Fractals

(type=lyapunov)

The Bifurcation fractal illustrates what happens in a simple population model as the growth rate increases. The Lyapunov fractal expands that model into two dimensions by letting the growth rate vary in a periodic fashion between two values. Each pair of growth rates is run through a logistic population model and a value called the Lyapunov Exponent is calculated for each pair and is plotted. The Lyapunov Exponent is calculated by adding up log | r - 2*r*x| over many cycles of the population model and dividing by the number of cycles. Negative Lyapunov exponents indicate a stable, periodic behavior and are plotted in color. Positive Lyapunov exponents indicate chaos (or a diverging model) and are colored black.

Order parameter.
Each possible periodic sequence yields a two dimensional space to explore. The Order parameter selects a sequence. The default value 0 represents the sequence ab which alternates between the two values of the growth parameter. On the screen, the a values run vertically and the b values run horizontally. Here is how to calculate the space parameter for any desired sequence. Take your sequence of a's and b's and arrange it so that it starts with at least 2 a's and ends with a b. It may be necessary to rotate the sequence or swap a's and b's. Strike the first a and the last b off the list and replace each remaining a with a 1 and each remaining b with a zero. Interpret this as a binary number and convert it into decimal.

An Example.
I like sonnets. A sonnet is a poem with fourteen lines that has the following rhyming sequence: abba abba abab cc. Ignoring the rhyming couplet at the end, let's calculate the Order parameter for this pattern.

```
  abbaabbaabab          doesn't start with at least 2 a's
  aabbaababab           rotate it
  1001101010            drop the first and last, replace with 0's and 1's
  512+64+32+8+2 = 618
```

An Order parameter of 618 gives the Lyapunov equivalent of a sonnet. "How do I make thee? Let me count the ways..."

Population Seed.
When two parts of a Lyapunov overlap, which spike overlaps which is strongly dependent on the initial value of the population model. Any changes from using a different starting value between 0 and 1 may be subtle. The values 0 and 1 are interpreted in a special manner. A Seed of 1 will choose a random number between 0 and 1 at the start of each pixel. A Seed of 0 will suppress resetting the seed value between pixels unless the population model diverges in which case a random seed will be used on the next pixel.

Filter Cycles.
Like the Bifurcation model, the Lyapunov allow you to set the number of cycles that will be run to allow the model to approach equilibrium before the lyapunov exponent calculation is begun. The default value of 0 uses one half of the iterations before beginning the calculation of the exponent.

Reference.
A.K. Dewdney, Mathematical Recreations, Scientific American, Sept. 1991

1.50 Magnetic Fractals

(type=magnet1m/.../magnet2j)

These fractals use formulae derived from the study of hierarchical lattices, in the context of magnetic renormalisation transformations. This kinda stuff is useful in an area of theoretical physics that deals with magnetic phase-transitions (predicting at which temperatures a given substance will be magnetic, or non-magnetic). In an attempt to clarify the results obtained for Real temperatures (the kind that you and I can feel), the study moved into the realm of Complex Numbers, aiming to spot Real phase-transitions by finding the intersections of lines representing Complex phase-transitions with the Real Axis. The first people to try this were two physicists called Yang and Lee, who found the situation a bit more complex than first expected, as the phase boundaries for Complex temperatures are (surprise!) fractals.

And that's all the technical (?) background you're getting here! For more details (are you SERIOUS ?!) read "The Beauty of Fractals". When you understand it all, you might like to rewrite this section, before you start your new job as a professor of theoretical physics...

In Fractint terms, the important bits of the above are "Fractals", "Complex Numbers", "Formulae", and "The Beauty of Fractals". Lifting the Formulae straight out of the Book and iterating them over the Complex plane (just like the Mandelbrot set) produces Fractals.

The formulae are a bit more complicated than the Z^2+C used for the Mandelbrot Set, that's all. They are :

```
      [     ] 2
      |  Z^2 + (C-1)   |
  MAGNET1 : | ------------- |
      |  2*Z + (C-2)   |
      [     ]

      [             ] 2
      |  Z^3 + 3*(C-1)*Z + (C-1)*(C-2)      |
  MAGNET2 : | ------------------------------------ |
      |  3*(Z^2) + 3*(C-2)*Z + (C-1)*(C-2) + 1  |
      [             ]
```

These aren't quite as horrific as they look (oh yeah ?!) as they only involve two variables (Z and C), but cubing things, doing division, and eventually squaring the result (all in Complex Numbers) don't exactly spell S-p-e-e-d ! These are NOT the fastest fractals in Fractint !

As you might expect, for both formulae there is a single related Mandelbrot-type set (magnet1m, magnet2m) and an infinite number of related Julia-type sets (magnet1j, magnet2j), with the usual toggle between the corresponding Ms and Js via the spacebar.

If you fancy delving into the Julia-types by hand, you will be prompted for the Real and Imaginary parts of the parameter denoted by C. The result is symmetrical about the Real axis (and therefore the initial image gets drawn in half the usual time) if you specify a value of Zero for the

Imaginary part of C.

Fractint Historical Note: Another complication (besides the formulae) in implementing these fractal types was that they all have a finite attractor (1.0 + 0.0i), as well as the usual one (Infinity). This fact spurred the development of Finite Attractor logic in Fractint. Without this code you can still generate these fractals, but you usually end up with a pretty boring image that is mostly deep blue "lake", courtesy of Fractint's standard Periodicity Logic. See Finite Attractors for more information on this aspect of Fractint internals.

(Thanks to Kevin Allen for Magnetic type documentation above).

1.51 L-Systems

(type=lsystem)

These fractals are constructed from line segments using rules specified in drawing commands. Starting with an initial string, the axiom, transformation rules are applied a specified number of times, to produce the final command string which is used to draw the image.

Like the type=formula fractals, this type requires a separate data file. A sample file, FRACTINT.L, is included with this distribution. When you select type lsystem, the current lsystem file is read and you are asked for the lsystem name you wish to run. Press <F6> at this point if you wish to use a different lsystem file. After selecting an lsystem, you are asked for one parameter - the "order", or number of times to execute all the transformation rules. It is wise to start with small orders, because the size of the substituted command string grows exponentially and it is very easy to exceed your resolution. (Higher orders take longer to generate too.) The command line options "lname=" and "lfile=" can be used to override the default file name and lsystem name.

Each L-System entry in the file contains a specification of the angle, the axiom, and the transformation rules. Each item must appear on its own line and each line must be less than 160 characters long.

The statement "angle n" sets the angle to 360/n degrees; n must be an integer greater than two and less than fifty.

"Axiom string" defines the axiom.

Transformation rules are specified as "a=string" and convert the single character 'a' into "string." If more than one rule is specified for a single character all of the strings will be added together. This allows specifying transformations longer than the 160 character limit. Transformation rules may operate on any characters except space, tab or '}'.

Any information after a ; (semi-colon) on a line is treated as a comment.

Here is a sample lsystem:

```
Dragon {    ; Name of lsystem, { indicates start
```

```
  Angle 8  ; Specify the angle increment to 45 degrees
  Axiom FX   ; Starting character string
  F=     ; First rule:  Delete 'F'
  y=+FX--FY+   ; Change 'y' into  "+fx--fy+"
  x=-FX++FY-   ; Similar transformation on 'x'
}    ; final } indicates end
```

The standard drawing commands are:

- F Draw forward
- G Move forward (without drawing)
- + Increase angle
- - Decrease angle
- | Try to turn 180 degrees. (If angle is odd, the turn will be the largest possible turn less than 180 degrees.)

These commands increment angle by the user specified angle value. They should be used when possible because they are fast. If greater flexibility is needed, use the following commands which keep a completely separate angle pointer which is specified in degrees.

- D Draw forward
- M Move forward
- nn Increase angle nn degrees
- /nn Decrease angle nn degrees

Color control:

- Cnn Select color nn
- <nn Increment color by nn
- >nn decrement color by nn

Advanced commands:

- ! Reverse directions (Switch meanings of +, - and \, /)
- @nnn Multiply line segment size by nnn
- nnn may be a plain number, or may be preceded by
  - I for inverse, or Q for square root.
  - (e.g. @IQ2 divides size by the square root of 2)
- [Push. Stores current angle and position on a stack
-] Pop. Return to location of last push

Other characters are perfectly legal in command strings. They are ignored for drawing purposes, but can be used to achieve complex translations.