Introduction to the Chaos library

Calling Sequence:

readlib(chaos)
function (args)

Description:

To use a chaos function, you must first execute the `readlib(chaos)` command. Most of the terminology used can be found in Devaney's undergraduate text, *Chaotic Dynamical Systems* (ISBN: 0-201-55406-2), Frame and Peak's *Chaos Under Control* (ISBN:0-7167-2429-4), or Barnsley's *Fractals Everywhere* (ISBN: 0-12-079062-9).

The functions available are:

Basic Dynamics

`Orbit(`*f:function,x:initial value,n:positive integer*`)`
- produces the first *n* exact values for the *f*–orbit of *x*. Returns a list.

`Orbitf(`*f:function,x:initial value,n:number of iterations*`)`
- computes the first n values of the orbit of x under f using floating point approximations. Returns a list.

`GraphicalAnalysis(`*f:function,a:lower bound,b:upper bound,* x_0*:seed,n:iterations*`)`
- displays a graphical analysis of *n* iterations of the orbit of x_0 under *f* over the interval $[a, b]$

`AnimatedGraphicalAnalysis(same arguments as GraphicalAnalysis)`
- performs a graphical analysis one step at a time and then animates the sequence. (The animation window operates like a VCR)

`FixedPointAnalysis(`*F:polynomial function*`)`
- tries to find and classifies all fixed points and 2 cycles for real valued polynomial function *F*

Bifurcation

`Bifurcation(`*f:function,* c_0*:lower limit,* c_1*:upper limit,hpoints:horizontal points, n:number of points,* x_0*:real number*`)`
- plots the bifurcation diagram of $f(x, c)$ for *c* in $[c_0, c_1]$ with increment $\frac{|c_1 - c_0|}{hpoints}$. It plots *n* points after throwing away the first 50 points. It uses an initial value of x_0 for the individual orbits.

`Bif3d(`x_0`,`y_0`,`x_1`,`y_1`,`*xgrid,ygrid,toss,keep*`)`
- Computes an extension of the bifurcation diagram in R^3 above the Mandelbrot set to the rectangular grid in the complex plane whose corners are $[x_0, y_0]$ and $[x_1, y_1]$ over a grid which is *xgrid* rows and *ygrid* columns. It throws away the first *toss* elements of each orbit and plots the next *keep* values using the absolute value of the complex values as the third coordinate.

Data Analysis

TimeSeriesPlot(*f*,*t*,*n*)
- plots the time series plot of the first *n* iterates of the *f*-orbit of *t*, i.e. plots the points [0,*t*],[1,*f*(*t*)],...,[*n*,$f^n(t)$].

ShowMixing(*f*,*t*,*n*)
-visually tests the mixing property of first *n* iterates the *f*-orbit of *t* by plotting the points [0.5,*t*],[0.5,*f*(*t*)],...,[0.5,$f^n(t)$]

FirstDelayPlot(*List*)
-plots the first delay plot of a list of data values, *List*, i.e. plots [x_0, x_1],[x_1, x_2],...,[x_{n-1}, x_n].

ClosePairsPlot(*DATA*,*tolerance*)
- plots the close-pairs plot of *DATA* (a list of real values) with tolerance *tolerance.*

ChaosGameTest(*DATA*)
- plays the 4 corner chaos game driven by *DATA*, which is a list of real values between 0 and 1.

CircleChaosGameTest(*DATA*)
- plays a variant of the chaos game driven by *DATA*, which is a list of real values between 0 and 1. Instead of using a fixed number of goal points and clustering the data into groups, the points on the unit circle are all goal points and are labeled from 0 to 1 CCW direction and each value is plotted by moving half way from the previous point to the point on the unit circle that is labeled by the current data value.

BarnsInt(*DATA*,*d*)
- returns an IFS object (see below) whose attractor is the Barnsley Fractal Interpolation IFS for the data points in *DATA*, which is a list of *n* ordered pairs [[x_1, y_1],...,[x_n, y_n]]. *d* is a list of *n*-1 parameters that influence the fractal dimension of the graph.

Particular Maps

Dbl(*x:real*)
- The doubling function

LogisticChart(λ*:real*,*n:pos. integer*)
- makes a nice table of the first *n* iterates of various orbits for logistic function $f_\lambda(x) = \lambda x (1-x)$

LogisticGrowthPlot(*c*,x_0,*n*,*L*)
-Plots the time series plot for the first *n* iterates of the *f*-orbit of x_0 logistic map $f(x) = c x (1-x)$ with nice formatting. If L='lines' then it connects the data points, otherwise it doesn't.

BoxLogisticGrowthPlot(*c)*
-Plots the time series plots for the first 50 iterates of the *f*-orbits of 0.1,0.2,...,0.8, and 0.9 under the logistic map $f(x) = c x (1-x)$ all on the same graph to see if they all have the same long term behavior or not.

ChaosGame(*PtList:List of Points*,*Start:starting point*,*NumPts:number of points*, *Ratio:ratio to move*)
- plays the chaos game starting with point *Start* and continuing by moving *Ratio* percent of the distance toward a randomly chosen point from *PtList*. It does this *NumPts* times. Points are input as lists, [*x*,*y*].

IFS's and AFFINE maps

In the following routines, we define two data types: AFFINE and IFS.
An IFS is a single AFFINE or a list of one or more AFFINE's.
An AFFINE represents an affine map of the plane and comes in four flavors:

```
affine(a,b,c,d,e,f)        - standard form
Affine(R,S,theta,phi,E,F)  - geometric form
affineC(α,β,γ)               - complex form
affineM(M,B)               - matrix form
```

These are inert forms representing the following maps:

`affine(a,b,c,d,e,f) <-> T(x,y)=(ax+by+e,cx+dy+f)`

`Affine(R,S,theta,phi,E,F) <-> T(x,y)=(` $R\cos\left(\frac{\pi\,\theta}{180}\right)x - S\sin\left(\frac{\pi\,\phi}{180}\right)y + E,$

$R\sin\left(\frac{\pi\,\theta}{180}\right)x + S\cos\left(\frac{\pi\,\phi}{180}\right)y + F$ `)`

`affineC(α,β,γ) <-> T(z)=` $\alpha z + \beta\overline{z} + \gamma$ where α, β, γ, z are complex numbers

`affineM(M,b) <->` $T(v) = Mv + b$ where M is a 2x2 matrix and b, v are column vectors

IFS routines

```
`convert\affine`(A::AFFINE)  - converts an AFFINE or IFS object to its affine()
form
`convert\Affine`(A::AFFINE)  - converts an AFFINE or IFS object to its Affine()
form
`convert\affineC`(A::AFFINE) - converts an AFFINE or IFS object to its affineC()
form
`convert\affineM`(A::AFFINE) - converts an AFFINE or IFS object to its affineM()
form
```

affineFromPoints($p_1, p_2, p_3, Tp_1, Tp_2, Tp_3$)
: - Computes the AFFINE object affine(*a,b,c,d,e,f*) for the unique affine map T(*x,y*)=(*ax+by+e,cx+dy+f*) which maps points p_1 to Tp_1, p_2 to Tp_2, and p_3 to Tp_3.

IFSFromList(*List*)
: -creates an IFS data structure from a list of *m* lists of the form [*R*, *S*, θ, φ, *E*, *F*]

Map(IFS)
: - converts the AFFINE or IFS object to the map T(*x,y*)=(*ax+by+e,cx+dy+f*)

Transform(IFS)
: - converts the AFFINE or IFS object to a transform which can be applied to plot objects

DrawDetIFS(*figure,IFS,n*)
: - plots the *n*th iteration of the deterministic method for *IFS* starting with *figure*

DrawIFS(*IFS,n*)
: - plots the attractor of IFS by the random iteration method using *n* points. (Note: don't use a value of *n* that is larger than about 30000)

RestrictedIFS(*IFS,n,S*)
: - plots the attractor of IFS by the random iteration method using *n* points, but does not plot any point whose address contains any of the

finite address lists given in the set *S*.

IFScurve(*t*,*IFS*)
- computes the point in the attractor of an *m* transformation *IFS* whose address begins with the digits in the base *m* expansion of *t*

DrawIFScurve(*IFS*,*n*)
- plots the attractor defined by IFS using *n* evenly spaced points in [0..1] for addresses

ContractionFactor(A)
-returns the contraction factors for the AFFINE map A (or a list of contraction factors if A is an IFS)

Built-in IFS's

The following IFS data types are built-in:

SierpinskiCarpet	SierpinskiTriangle	SierpinskiTriangle2
KochCurve	CantorSet	

In addition IFS's can be quickly produced with the following routine:

GridIFS()
- this routine takes n^2 arguments, each of which is from the set $\{Lt, Rt, Up, Dn, -Lt, -Rt, -Up, -Dn, None\}$. It returns the following IFS: Divide the unit square [0..1]x[0..1] into an *n*x*n* grid. Number the grid squared from 1 to n^2 starting in the lower left corner and moving from left to right and then bottom to top. The *i*th argument describes an affine transformation which maps the unit square into the *i*th grid cell in a manner analogous to one of the symmetry operations in the dihedral group of the square (e.g. Lt means it is rotated 90 degrees CCW, Rt means 90deg CW, -Up is a reflection along a vertical axis, -Lt is reflection along the vertical axis followed by 90deg CCW rotation, etc). If the argument is none, there is no affine map associated with that grid cell. For example, the right Sierpinski Triangle can be produced with GridIFS (Up,Up,Up,none)

HeeBGB()
- an alias for GridIFS() used in my classes to denote a method for coloring Grid-Based (hence the GB) IFS fractals by hand. Syntax is exactly the same as GridIFS().

GB(*n*,x_1,...,x_k)
- this routine is a shorthand for inputting GridIFS() fractals that only involve the *Up* and *None* arguments. The argument *n* is the same as *n* in the description of GridIFS() and x_1,...,x_k are positive integers less than or equal to n^2 which indicate the locations of the *None*'*s* on the grid described in GridIFS().

Built-in starting figures

The following plot structures are built-in, and can be used as the starting figure for plotting the iterations of the deterministic method for an IFS (see DrawDetIFS()).

MrFace	MrGrid	MrPoint	MrLine
MrSquare	MrTriangle	MrsTriangle	

Directed Segment Replacement Fractals (aka stick figures)

In the following routines, we define a data type: STICK
A STICK is an inert Maple object of one of the following two forms:

```
     seg([a,b],[c,d])    - represents a line segment with end points [a,b] and
[c,d]
     dseg([a,b],[c,d])   - represents a directed line segment from [a,b] to [c,d]
```

A set of STICKS is called a STICK FIGURE.

Directed Segment Replacement routines

DrawSticks(s)
- Plots the STICK FIGURE *s*. If the optional argument 'showdsegs' is given after *s*, the directed segments are drawn with an arrow at their midpoint indicating their direction.

DrawSF(*seed*,*Replace*,*n*)
- Let $x_0,...,x_n$ be a sequence of STICK FIGURES defined as follows. x_0=*seed*. For 0<$i \le n$ define x_i to be the stick figure obtained by replacing each of the dseg()'s in x_i with the appropriately transformed STICK FIGURE *Replace*. In particular, if dseg([a,b],[c,d]) is any directed segment in x_{i-1} and [x,y] is any endpoint of any STICK in *Replace*, then x_i will contain a corresponding STICK (of the same type as the one in *Replace*) having an end point at $[(c-a)\,x+(b-d)\,y+a,\,(d-b)\,x+(c-a)\,y+b]$. The original directed segment is replaced while non-directed seg()'s in x_{i-1} are not transformed and are simply passed along to x_i. The DrawSF() command plots x_n. If the optional argument 'showdsegs' is given after *n*, the directed segments are drawn with an arrow at their midpoint indicating their direction. (This is actually a lot simpler than it sounds!)

Built-in starting stick figures

The following plot structures are built-in, and can be used as the starting figure (seed) for DrawSF().

```
      MrDseg      MrDtriangle
```

Mandelbrot and Julia type fractals

JuliaFractal(*Fx,Fy,a..b,c..d,options*)
- computes a Julia-set type fractal for a function *F*:*C*->*C* where *C* is the complex plane. The arguments are as follows:

Fx - the real part of *F*
Fy - the real part of *F*
a..b - an optional range indicating the horizontal range to be displayed (the default is -2..2)
c*..d* - an optional range indicating the vertical range to be displayed (the default is -2..2)

Options:
The remaining options given as equations of the form option = value. The

following options are supported:

maxiter - a positive integer indicating the maximum number of iterations to compute for each point before declaring that point to be in the filled in Julia set associated with F (default is 500)

radius - a positive real indicating the escape radius (default is 2)

grid - a list, [n,m], of integers indicating the number of grid points horizontally and vertically to use (default is [100,100])

scheme - an integer from 1..6 indicating which color scheme to use (default is 1)

rate - a number from 1 to maxiter indicating the rate at which colors should change. Low numbers will produce a striped coloration while higher values produce a gradual transition (default is 25)

timer - a boolean indicating whether the routine should report how long it took to compute the fractal (default is *false*)

Any additional arguments are passed along to plots[display] before rendering.

MandelFractal(*Fx,Fy,a..b,c..d,options*)

- computes a Mandelbrot set type fractal for a family of functions *F*:*C*->*C* where *C* is the complex plane. The arguments are as follows:

Fx - the real part of *F* (must be a proc(x,y,c0,c1) returning a real number)
(default value is *(x,y,c0,c1)->x*x-y*y+c0*)
Fy - the real part of *F* (must be a proc(x,y,c0,c1) returning a real number)
(default value is *(x,y,c0,c1)->2*x*y+c1*)
a..b - an optional range indicating the horizontal range to be displayed
(the default is -2..2)
c*..d* - an optional range indicating the vertical range to be displayed
(the default is -2..2)

Options:
The remaining options given as equations of the form option = value. The following options are supported:

maxiter - a positive integer indicating the maximum number of iterations to compute for each point before declaring that point to be in the Mandelbrot set associated with F (default is 500)

radius - a positive real indicating the escape radius (default is 2)

grid - a list, [n,m], of integers indicating the number of grid points horizontally and vertically to use (default is [100,100])

scheme - an integer from 1..6 indicating which color scheme to use (default is 1)

rate - a number from 1 to maxiter indicating the rate at which colors should change. Low numbers will produce a striped coloration while higher values produce a gradual transition (default is 25)

timer - a boolean indicating whether the routine should report how long it took to compute the fractal (default is *false*)

Any additional arguments are passed along to plots[display] before rendering.

MakePalette(*n*,*rate*,*scheme*)
- creates a color palette with *n* colors and period *rate.* There are six schemes available numbered 1..6.

ViewPalette(*palette*)
- displays the color palettes that are output by MakePalette()

ComplexToRealFunc(*F*)
- returns the real and imaginary parts of a complex function *F*:*C*->*C*

MandComplexToRealFunc(*F*)
- returns the real and imaginary parts of a complex family of functions *F*:*C*->*C* parameterized by c0+c1*I

QuadraticFunc(*c*)
- returns the map z->z^2+c where c is a complex number

JuliaSet(*c*)
- shorthand for JuliaFractal(ComplexToRealFunc(QuadraticFunc(*c*))). Additional args are passed along to JuliaFractal().

MandelbrotSet()
- shorthand for MandelbrotFractal(). Additional args are passed along to MandelbrotFractal ().

FRACTRAN

Fractran(*List*)
-returns the Fractran function defined by the list of rational numbers, *List*

IsTwoPower(n)
-returns true if n is an integer power of 2, and false otherwise

TwoExponent(2^n)
-returns n

Post's Tag Problem

TAG(*string*)
-Post's TAG function for strings of a's and b's

Other Useful Utilities

Table()
- Takes lists as args and prints them in a table.

BaseN(*t*,*n*)
-Converts a real number *t* in [0..1] to base *n*... returns a list of *Digits* digits.

FatPoint(*x:real*,*y:real*,*n:pos. integer*,*siz:pos. real*,*col:color constant*)
- plots a big fat point at (x,y) as a regular polygon with *n* sides and radius *siz* with color *col.* Used to make special points more visible in a plot.

Examples

```
restart
with(chaos)
f := x -> x^2 - 2; Q := (x, c) -> x^2 + c; L := (x, c) -> c*x*(1 - x)
Orbit(f, 1/2, 10)
Orb := Orbitf(f, 1/2, 6)
Table(Orb, numbered)
Orb := Orbit(x -> Q(x, -0.78 + 0.04*I), 0, 10):
Table([Orb, opts(digits = 4)], map(abs, Orb))
GraphicalAnalysis(f, -2, 2, 0.65, 250)
AnimatedGraphicalAnalysis(f, -2, 2, 0.65, 50)
FixedPointAnalysis(x -> x^2 - 1)
Bifurcation(Q, -2, 0.25, 500, 100, 0)
Bif3d(-2, -2, 2, 2, 40, 40, 150, 10)
TimeSeriesPlot(x -> x^2 - 0.75, 0, 100)
ShowMixing(x -> 3.95*x*(1 - x), 0.5, 200)
n := 1000: Random := rand(1..10^11): List1 := [seq(evalf(Random()/10^11), i = 1..n)]:
List2 := Orbit(x -> 3.99*x*(1 - x), 0.328763415, n - 1):
# output omitted: Table(List1,List2,numbered):
FirstDelayPlot(List1)
FirstDelayPlot(List2)
ClosePairsPlot(List1[1..100], 0.05)
ClosePairsPlot(List2[1..100], 0.05)
ChaosGameTest(List1)
ChaosGameTest(List2)
CircleChaosGameTest(List1)
CircleChaosGameTest(List2)
```

```
mb := BarnsInt([[1, 2], [2, 3], [3, 1], [4, 2]], [0.4, 0.4, 0.4])
IFSCurve(0, mb); IFSCurve(1/3, mb); IFSCurve(2/3, mb); IFSCurve(1, mb)
IFSCurve(0.5, mb)
DrawIFSCurve(mb, 400)
plot(Dbl(x), x = 0..1, discont = true)
LogisticChart(3.2, 25):
LogisticGrowthPlot(3.5, 0.3, 100, lines)
BoxLogisticGrowthPlot(3.5)
ChaosGame([[0, 0], [0, 1], [1, 0]], [0, 0], 10000, 0.5)
ChaosGame([[0, 0], [1, 1], [1, 0], [0, 1]], [0.3, 0.3], 10000, 0.55)
T := affineFromPoints([0, 0], [1, 0], [0, 1], [0, 1], [0, 1/2], [1/2, 1/2])
convert(T, Affine)
convert(T, affineC)
convert(T, affineM)
Map(T)
Mapf(T)
P := plot(sin(x), x = -6..6): P
display(Transform(T)(P))
ContractionFactor(T)
MyIFS := GridIFS(-Up, none, none, -Lt, none, Up, Lt, none, none, Rt, Dn, none, -Rt, none,
    none, -Dn)
Map(MyIFS)
Map(MyIFS)[1]
Map(MyIFS)[1](2, 3)
convert(MyIFS, affine)
convert(MyIFS, affineC)
convert(MyIFS, affineM)
ContractionFactor(MyIFS)
DrawDetIFS(MrFace, MyIFS, 1)
DrawDetIFS(MrFace, MyIFS, 2)
DrawIFS(MyIFS, 20000)
DrawIFS(GridIFS(Up, Lt, -Dn, n), 30000)
DrawIFS(HeeBGB(Up, Lt, -Dn, n), 30000)
DrawIFS(GridIFS(Up, n, Up, n, Up, n, Up, n, Up), 30000)
DrawIFS(GB(3, 2, 4, 6, 8), 30000)
RestrictedIFS(GridIFS(Up, Up, Up, Up), 30000, {[0, 0], [1, 1], [2, 2], [3, 3]})
NiceCurveIFS := IFSFromList([[-1/2, 1/2, -120, -120, 0, 0], [1/2, 1/2, 0, 0, 1/4, sqrt(3)/4],
    [-1/2, 1/2, 120, 120, 3/4, sqrt(3)/4]])
IFSCurve(0.5, NiceCurveIFS)
```

```
DrawIFSCurve(NiceCurveIFS, 3^6)
DrawIFS(SierpinskiCarpet, 30000)
DrawIFS(SierpinskiTriangle, 20000)
DrawIFS(SierpinskiTriangle2, 20000)
DrawIFS(KochCurve, 10000)
DrawIFS(CantorSet, 5000)
DrawDetIFS(MrFace, GridIFS(Up, none, −Up, none, Lt, none, Rt, none, Lt), 1, scaling
    = constrained)
DrawDetIFS(MrGrid, GridIFS(Up, none, −Up, none, Lt, none, Rt, none, Lt), 1, scaling
    = constrained)
DrawDetIFS(MrPoint, GridIFS(Up, none, −Up, none, Lt, none, Rt, none, Lt), 1, axes = none,
    scaling = constrained)
DrawDetIFS(MrLine, GridIFS(Up, none, −Up, none, Lt, none, Rt, none, Lt), 1, axes = none,
    scaling = constrained)
DrawDetIFS(MrSquare, GridIFS(Up, none, −Up, none, Lt, none, Rt, none, Lt), 1, scaling
    = constrained)
DrawDetIFS(MrTriangle, GridIFS(Up, none, −Up, none, Lt, none, Rt, none, Lt), 1, scaling
    = constrained)
DrawDetIFS(MrsTriangle, GridIFS(Up, none, −Up, none, Lt, none, Rt, none, Lt), 1, scaling
    = constrained)
KochSnowflake := {dseg([0, 0], [1/3, 0]), dseg([1/2, sqrt(3)/6], [2/3, 0]), dseg([1/3, 0], [1/2,
    sqrt(3)/6]), dseg([2/3, 0], [1, 0])}
DrawSticks(KochSnowflake)
DrawSticks(KochSnowflake, showdsegs)
DrawSticks(MrDseg, axes = boxed, view = [−0.1..1.1, −0.5..0.5])
DrawSticks(MrDtriangle, axes = boxed)
DrawSF(MrDtriangle, KochSnowflake, 1, showdsegs)
DrawSF(MrDtriangle, KochSnowflake, 2, showdsegs)
DrawSF(MrDtriangle, KochSnowflake, 5)
Tree := {dseg([0, 2], [1/2, 2.4]), dseg([1/2, 2.4], [1, 2]), seg([0, 0], [0, 2]), seg([1, 0], [1,
    2])}:
Seed := {dseg([0, 0], [1, 0]), seg([0, 0], [1, 0])}: DrawSF(Seed, Tree, 9)
JuliaSet(−1)
JuliaSet(−1, scheme = 6, rate = 2)
JuliaSet(−1, 1.3..1.75, −0.15..0.15, maxiter = 50, grid = [50, 50], scheme = 2, timer = true)
JuliaSet(−1, 1.3..1.75, −0.15..0.15, maxiter = 400, grid = [150, 150], scheme = 2, timer = true)
JuliaSet(−0.7519 + 0.03523 I)
ComplexToRealFunc(z ↦ sin(z))
JuliaFractal(ComplexToRealFunc(z ↦ sin(z)), −6..6, −6..6, radius = 20, maxiter = 200, grid
```

$= [150, 150], scheme = 6, rate = 15, timer = true)$

$MandelbrotSet( )$

$MandelbrotSet(timer = true)$

$MandelbrotSet(-0.7519 .. -0.75158, 0.03523 .. 0.035563, maxiter = 10000, grid = [300, 300], scheme = 1, rate = 1000, timer = true, axes = normal)$

$MandComplexToRealFunc(z \mapsto \sin(z) + c0 + \mathrm{I} \cdot c1)$

$MandelFractal(MandComplexToRealFunc(z \mapsto \sin(z) + c0 + \mathrm{I} \cdot c1), -6..6, -6..6, radius = 20, maxiter = 400, grid = [150, 150], scheme = 6, timer = true, axes = framed)$

$MakePalette(10, 10, 1)$

$ViewPalette(MakePalette(10, 10, 1))$

$$CollatzGame := \left[\frac{1}{11}, \frac{136}{15}, \frac{5}{17}, \frac{4}{5}, \frac{26}{21}, \frac{7}{13}, \frac{1}{7}, \frac{33}{4}, \frac{5}{2}, 7\right]$$

$Orbit(Fractran(CollatzGame), 2^3, 59)$

$map(ifactor, select(IsTwoPower, \%))$

$map(TwoExponent, select(IsTwoPower, Orbit(Fractran(CollatzGame), 2^3, 59)))$

$$BaseN\left(\frac{1}{5}, 4\right)$$

The next three commands are not part of the chaos library but are useful for saving fractals images to postscript or jpeg files instead of displaying them on the screen.

```
#plotsetup(ps,plotoptions="colour=rgb,width=8in,height=6in",plotoutput="C:/testpic.ps");
#plotsetup(jpeg,plotoptions="width=800,height=600",plotoutput="C:/testpic.jpg");
#plotsetup(default,plotoutput=default);
```