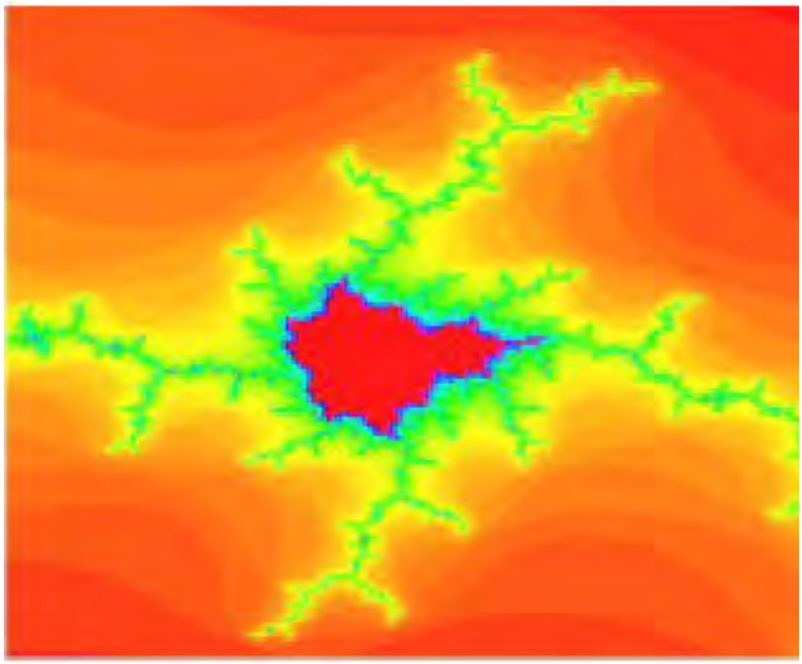
Chaotic Rendezvous:

Introductory Chaos Experimentation Using Maple Software and a Desktop Computer

by Thomas E. Oberst



Chaotic Rendezvous: Introductory Chaos Experimentation Using Maple Software and a Desktop Computer

Thomas E. Oberst

Duquesne University
Department of Physics
Pittsburgh, PA 15282

Chapter 10

Julia Sets

In this chapter we extend our analysis of dynamical functions to include complex functions. When we iterate certain complex functions we will find that the orbits of some initial seeds remain bounded while most others escape to infinity. The set of initial seeds whose orbits remain bounded constitute a special set known as the *filled Julia set* of that complex function. The boundary of the filled Julia set is simply called the *Julia set*. While their definitions are quite simple, the geometry of a Julia set or a filled Julia set which has been plotted in the complex plane can be very complicated. The great majority of Julia sets are fractals. This is not entirely surprising, however, since it can be shown that all of the chaotic behavior of a complex function takes place on the Julia set of that function.[1] Julia sets provide one of the major links between chaotic nonlinear dynamical systems and fractals.

The true beauty of Julia sets lies not only in their mathematical workings but also in their aesthetic spectacle. In this chapter and the experiments that follow we try to stay mindful of both. For many people, Julia sets are beautiful and amazing pictures. For us, the beauty and amazement are greatly multiplied by an understanding of the simple yet powerful mathematics "behind the scenes."

Before we begin, it would probably be useful to briefly review some concepts from complex variable theory. Complex functions look just like ordinary functions. For example,

$$Q_c(z) = z^2 + c \tag{10.1}$$

represents a family of complex quadratic functions, even though it exactly resembles in form the family of real quadratic functions, $Q_c(x) = x^2 + c$. The first difference is that c in $Q_c(x)$ represents a real number, while c in $Q_c(z)$ represents a complex number. Recall that a complex number has the form $c = a + ib$ (in Cartesian coordinates), where a and b are real numbers and $i = \sqrt{-1}$. a is called the *real part* of c, and b is called the *imaginary part* of c. The second major difference between the complex quadratic function and the real quadratic function is that $Q_c(z)$ must take a complex variable z as its argument while $Q_c(x)$ takes a real variable x as its argument (by saying "real variable," I mean a variable that represents one real number). Recall that complex variables are expressed in the form

[1] See Devaney, 1992, page 221.

$z = x + iy$ (in Cartesian coordinates), where x and y are real variables and, again, $i = \sqrt{-1}$. Similar to the case for the complex number c, x is called the real part of z and y is called the imaginary part of z.

Hence, $Q_c(z)$ could also be written in the form

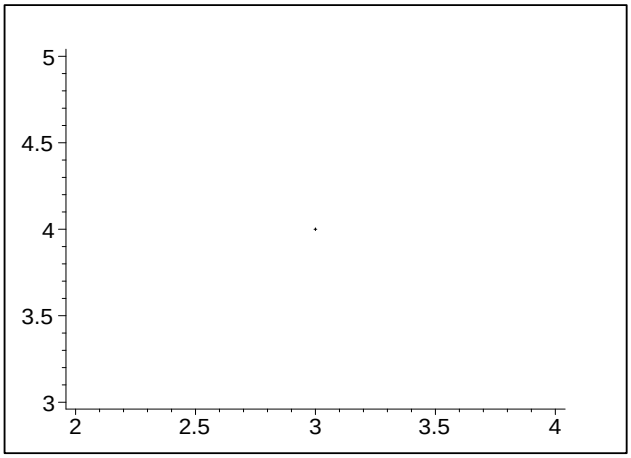
$$Q_c(z) = (x + iy)^2 + (a + ib) \tag{10.2}$$

or

$$\begin{aligned} Q_c(z) &= x^2 - y^2 + 2ixy + a + ib \\ Q_c(z) &= (x^2 - y^2 + a) + i(2xy + b) \end{aligned} \tag{10.3}$$

where $a, b \in \Re$ ($\Re$ is the set of all reals) and x and y are variables which represent numbers in $\Re$. Here we see that $\Re(Q_c(z)) = (x^2 - y^2 + a)$ is the real part of the function and $\Im(Q_c(z)) = (2xy + b)$ is the imaginary part of the function. Note that if we take only those members of $Q_c(z)$ for which $b = 0$ and agree to evaluate those functions only for complex variables in which $y = 0$, then this family of complex quadratic functions reduces to the family of real quadratic functions $Q_c(x)$.

Complex numbers can be plotted in the *complex plane*. The complex plane looks very much like a Cartesian x- and y-axis plane. To plot a complex number $a + ib$ in the complex plane, we plot a along to the x-axis of the plane, and b along to the y-axis of the plane. The x-axis is thus dubbed the "real axis" and the y-axis the "imaginary" axis. The x-axis in the complex plane is equivalent to the set of all real numbers. As an example, let's plot the complex number $3 + 4i$ on the complex plane:



The *modulus* of a complex number is the distance from that number to the origin in the complex plane. The modulus of a complex number z is denoted by $|z|$. Using the Pythagorean theorem, we can see quite easily that $|z| = |x + iy| = \sqrt{x^2 + y^2}$. Therefore,

we can say that the modulus of a complex number is equal to the sum of the square of its real part and the square of its imaginary part.

With that, our quick review of complex variable theory is complete and we are ready to introduce the definition of a Julia set.

Definition 10.1 *The* filled Julia set *of a complex function f is the set of all complex initial seeds whose orbits are bounded. The* Julia set *of a complex function f is the boundary of the filled Julia set.*

A bounded set is one which has a largest (non-infinite) member. A filled Julia set is denoted with a capital K and a Julia set is denoted with a capital J.

We are just about ready to compute and plot our first Julia set. Let us continue using Q_c as our example. To determine the filled Julia set of Q_c, we should calculate the orbits of all the points in the complex plane and note whether or not each orbit goes to infinity. But we surely can't check *every* point in the plane, since the plane is dense and there exist infinitely many points on it. What we need to do is take a random sample of a large number of points from the plane. We will use Maple's random number generating function, `rand`, which we used previously in Chapter 9. Recall that `rand` takes a range of integers as its argument and outputs a randomly selected integer from that range:

```
> f:=rand(0...1000000):
> f();
                                  9616
> f();
                                 751277
> f();
                                 83785
```

The question is, how large should we make our range? If we randomly choose complex numbers from a grid with length 1 million and width 1 million in the complex plane, can we be sure that part of the Julia set (and perhaps an interesting part) doesn't lie outside this grid? The answer is yes.

Theorem 10.1 (The Escape Criterion) *If $|z| \geq max(|c|, 2)$, then $|Q_c^n(z)| \to \infty$ as $n \to \infty$.*

The Escape Criterion tells us that the orbit of any initial seed $z = x + iy$ whose modulus $|z| = \sqrt{x^2 + y^2}$ is larger than the greater of 2 or the modulus of c, $|c| = \sqrt{a^2 + b^2}$, will blow up. Therefore, we can limit ourselves on the complex plane to a circle with a radius equal to the larger of 2 or $\sqrt{a^2 + b^2}$.[2]

[2]For a proof of the Escape Criterion, see Devaney, 1992, page 228.

With this simplification, however, we encounter a new complication: the `rand` function can only select integers. This seems to suggest that the `rand` function will be useless if we wish to sample a large number of points within a circle of radius 2. But we can "fool" Maple by doing the following: multiply 2 by a large power of 10, such as 10^{10}. Next, we use `rand` to select a random integer from between 0 and 2×10^{10}. Finally, we divide this integer by 10^{10}. The result will be a randomly chosen real number between 0 and 2 with accuracy to the tenth decimal place.

```
> R:=evalf(rand(0..2*10^10)/10^10):
> R();
                                .9392673688
> R();
                                .5428510947
> R();
                                1.260060897
```

It is important to remember that sometimes we will also be selecting randomly chosen points from a circle of radius $\sqrt{a^2+b^2}$ in the case when $\sqrt{a^2+b^2} > 2$. Since a and b are permitted to be any real numbers, it is possible that, for instance, $\sqrt{a^2+b^2} = 3.141592653589793$. In this case, multiplying by 10^{10} will leave us with the number 31415926535.89793. If we were to subsequently ask Maple to perform the function `rand`(0..31415926535.89793), we would receive an error message, since `rand` can only select from a range of integers. Therefore, we must always make sure we multiply by a large enough power of 10 so as to convert the decimal value $\sqrt{a^2+b^2}$ into an integer. As a safeguard, we can use the `trunc` command, which will round a given number down to the nearest integer less than that number.

```
> R:=evalf(rand(0..3.141592653589793*10^10)/10^10):
Error, (in rand) invalid range
> R:=evalf(rand(0..trunc(3.141592653589793*10^10))/10^10):
> R();
                                .8575733856
> R();
                                2.082260103
> R();
                                2.992705730
```

Once we randomly select a point from the complex plane that could potentially belong to the filled Julia set of Q_c, all we need to do is compute the orbit of that point and see whether it remains bounded within a circle of radius equal to $\max(|c|, 2)$. We know that any orbit which wonders outside of $\max(|c|, 2)$ will diverge to ∞ and will not belong to the filled Julia set. The only way to be absolutely sure that a point z_0 belongs to the filled Julia set is to iterate $Q_c(z_0)$ infinitely many times. This is an impossibility, even

in a lifetime of 80 years. The best we can do is pick a sufficiently large number of iterations, N, and iterate $Q_c(z_0)$ N times. If, after N iterations, the value of the orbit is still bounded by $\max(|c|, 2)$, then we assume that z_0 is in the filled Julia set. This approach is not foolproof, of course, since it is possible that z_0 will eventually leave the circle of radius $\max(|c|, 2)$ only after a number of iterations greater than N. Consequently, the larger we make N, the more accurate a picture of the filled Julia set of Q_c we will get. However, a filled Julia set with a large number of iterations will require a greater amount of time to plot.

Below is a crude algorithm I have written for computing and plotting a filled Julia set for a given complex quadratic function $Q_c(z) = z^2 + c$. I have named the procedure `julia_bw` (the "bw" indicating that it produces black and white plots). `julia_bw` allows the user to have control over the values of a and b, over the total number of initial seeds $z_0 = x + iy$ to sample ("numpoints"), over the maximum number of iterations to perform for each initial seed ("maxit"), and also over the number of decimal places to be used when randomly selecting values of x and y for the initial seeds ("decplaces"). The "decplaces" argument can be thought of as the spacing between randomly selected points. For example, setting "decplaces"=10 would allow points spaced as close as 10^{-10} apart to be chosen—as in the examples above with the `rand` command. These five parameters, once specified by the user, remain fixed while the algorithm is processed.

```
> julia_bw:=proc(a, b, numpoints, maxit, decplaces)
> local X, Y, juliaset, R, i, j:
> X:=array(0..maxit+1):  Y:=array(0..maxit+1):
> juliaset:=array(0..numpoints):
> juliaset[0]:={}:
> R:=evalf(rand(-trunc(max(2,sqrt(a^2+b^2))*10^decplaces)..trunc(max(2,
> sqrt(a^2+b^2))*10^decplaces))/10^decplaces);
> for i from 1 to numpoints do
> X[0]:=R():  Y[0]:=R():
> for j from 0 while sqrt(X[j]^2+Y[j]^2)<max(2,sqrt(a^2+b^2)) and
> j<=maxit do
> X[j+1]:=X[j]^2-Y[j]^2+a;
> Y[j+1]:=2*X[j]*Y[j]+b;
> od:
> if j>maxit then juliaset[i]:=(juliaset[i-1] union
> {[X[0],Y[0]]}):
> else juliaset[i]:=juliaset[i-1]:
> end if:
> od:
> with(plots):
```

```
> pointplot(juliaset[numpoints],
> view=[-max(2,sqrt(a^2+b^2))..max(2,sqrt(a^2+b^2)),-max(2,sqrt(a^2+b^2)
)..max(2,sqrt(a^2+b^2))], symbolsize=10,
> symbol=diamond,
> scaling=constrained, color=black, axes=frame):
> end:
```

In the procedure, we define arrays X and Y to hold the iterative values of the real and imaginary parts of our complex iteration variable z, respectively. We then define the zeroth iterates of the arrays X and Y, X_0 and Y_0, within a `for` loop so that each time through the loop we start with new values of X_0 and Y_0—and hence a new variable z from the complex plane. The Escape Criterion tells us that we would be wise to choose these initial points X_0 and Y_0 from within a circle of radius equal to $\max(\sqrt{a^2+b^2}, 2)$. Therefore, with each round of the `for` loop we define X_0 and Y_0 using the `rand` generating function (as explained above) to choose random real numbers from within a circle in the complex plane of radius equal to $\max(\sqrt{a^2+b^2}, 2)$. We perform the loop a number of times equal to our "numpoints" specification in the argument of the command `julia_bw`. Thus, "numpoints" is the number of initial seeds in the plane which we will sample, but not necessarily the number of points that will be plotted. Nested within this loop, we use a second `for` loop to iterate the function $Q_c(z) = z^2 + c$ with the initial seed $z_0 = X_0 + iY_0$ up to a number equal to our "maxit" specification in the argument of the command `julia_bw`. It is quite possible, however, that the orbit of a given initial seed z_0 will escape from the circle of radius equal to $\max(\sqrt{a^2+b^2}, 2)$ before a "maxit" number of iterations has been performed. We therefore include the stipulation, `while` $\sqrt{X_j^2 + Y_j^2} < \max(2, \sqrt{a^2+b^2})$, which will stop the loop if z_0 does leave this circle. Although Maple is capable of doing complex arithmetic, to speed up the iteration process we break down the computation so that it includes only real numbers. We do this as follows.

We know that

$$Q_c(z) = z^2 + c \tag{10.4}$$

Or,

$$Q_{a+ib}(x+iy) = (x^2 - y^2 + a) + i(2xy + b) \tag{10.5}$$

as we saw in Equation 10.3. Since i is constant, Q really only depends on the variables a and b, and is really only a function of the variables x and y. We can therefore rewrite this as

$$Q_{a,b}(x,y) = (x^2 - y^2 + a) + i(2xy + b) \tag{10.6}$$

So if $z_0 = x + iy$, then $z_1 = (x^2 - y^2 + a) + i(2xy + b)$. Because we are using arrays X and Y to hold the values of x and y rather than z, let's restate this as follows: if $z_0 = x_0 + iy_0$, then $z_1 = (x_0^2 - y_0^2 + a) + i(2x_0y_0 + b)$. Therefore,

$$x_1 = x_0^2 - y_0^2 + a \tag{10.7}$$

and

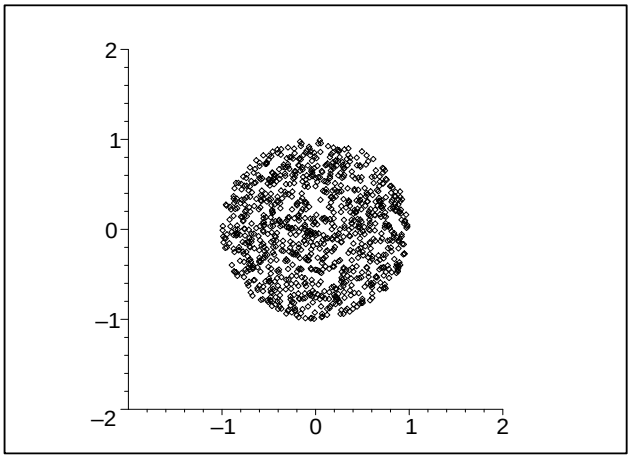
$$y_1 = 2x_0y_0 + b \tag{10.8}$$

Hence, we can use the iterative formulas 10.7 and 10.8 within the loop to speed up the algorithm.

After this inner loop iterates the function for one given X_0 and Y_0 initial seed pair (i.e. for one round of the outer loop), we use an `if-then` statement to either include the point $z_0 = X_0 + iY_0$ in our Julia set or to exclude it. If the point belongs to the filled Julia set, we place it in the array "juliaset" using Maple's `union` command for set manipulation. If the point does not belong to the filled Julia set, we set "juliaset" equal to itself. Finally, the output of our procedure is a `pointplot` of all of the ordered pairs (x, y) in our final "juliaset." The view is adjusted so that it will include just the circle of radius $\max(\sqrt{a^2 + b^2}, 2)$. The other options of `pointplot` are optional.

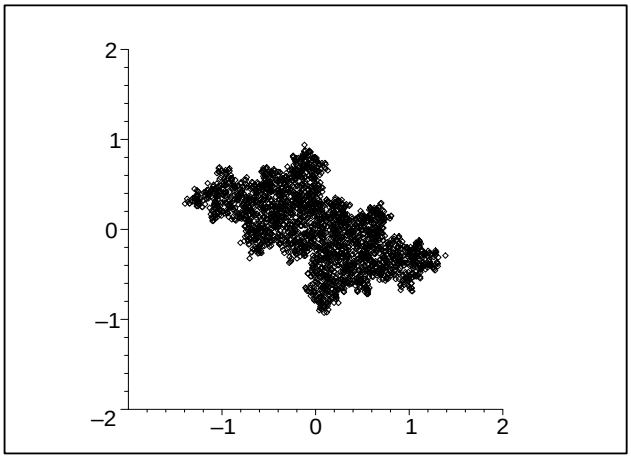
Here are a few examples of some filled Julia sets. In the first example we set $a = b = 0$ so that our function becomes $Q_0(z) = z^2$. From this, we should expect that any z for which $|z| > 1$ would blow up to infinity. Hence, the filled Julia set should be a filled unit circle. Because it shouldn't be too difficult to see a unit circle, we set our number of points to a rather low number, such as 5000. We also don't need a lot a definition around the border of the circle, so we can set our maximum number of iterations equal to 15. A decimal place accuracy of 10 digits should be sufficient:

```
> julia_bw(0, 0, 5000, 15, 10);
```

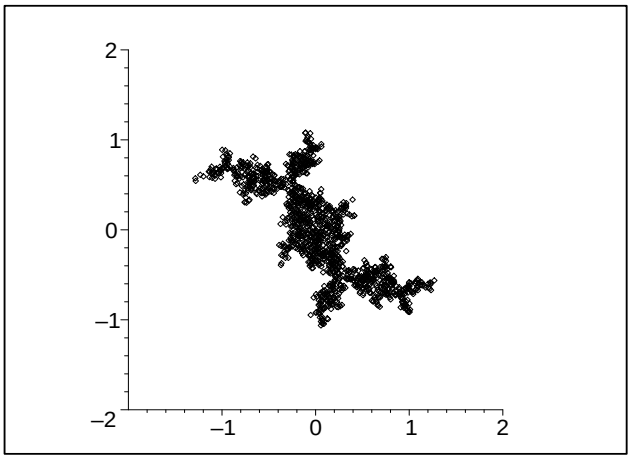


In the other examples that follow we have randomly chosen a few interesting values of a and b. Because these filled Julia sets may have complicated shapes, we increase our number of points to 20,000 to get a sharper image.

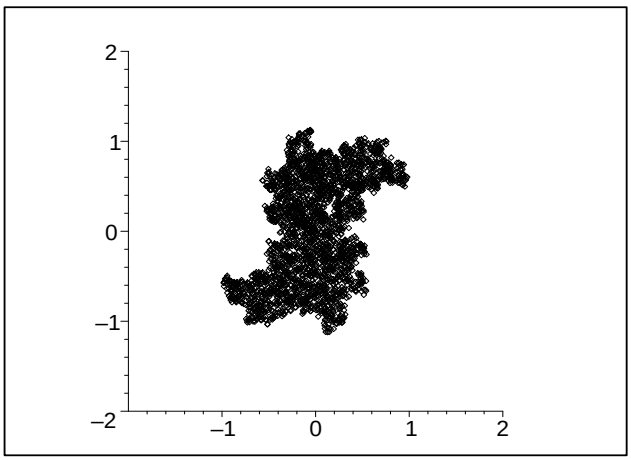
```
> julia_bw(-.5, .5, 20000, 15, 10);
```




```
> julia_bw(-.1, .8, 20000, 15, 10);
```

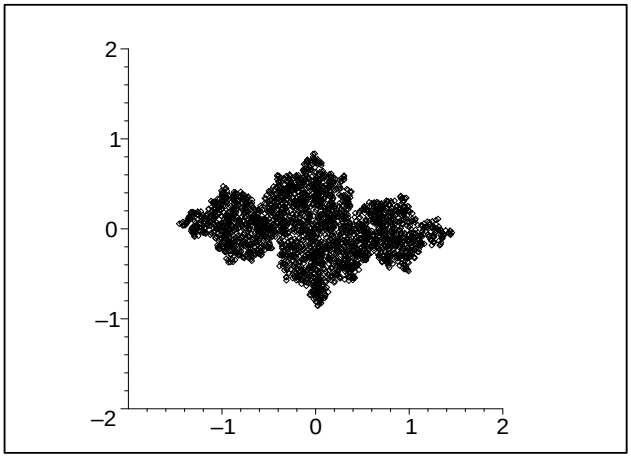



```
> julia_bw(.3, -.4, 20000, 15, 10);
```



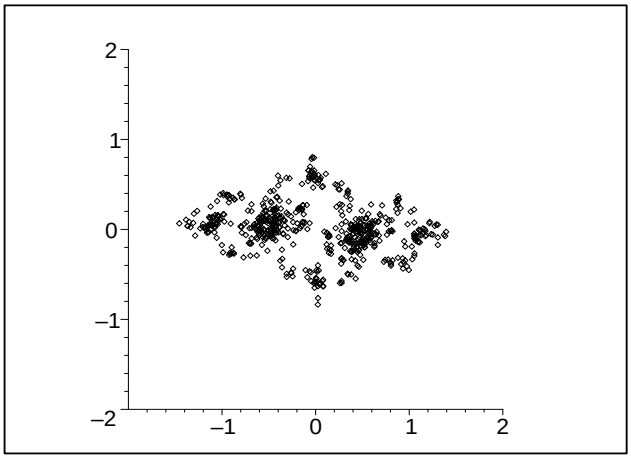
If a sharper, denser image is desired, the number of sampled points can be increased above 20,000 and the number of iterations can be increased above 15. But the larger the number of points and number of iterations, the longer it takes for the algorithm to calculate the filled Julia set. Ideally, we must find the right balance between a number of points and iterations which will give reasonably accurate results and the amount of time we have. The number of points ("numpoints") controls the density of the image. The number of iterations ("maxit") controls the accuracy and sharpness of the boundary of the filled Julia set. The reason for this is that if a given point remains in the filled Julia set for a high number of iterations, it is likely that the point does indeed belong to the filled Julia set. If we only perform five iterations, for instance, many points which still appear to be in the filled Julia set probably don't belong there, and would leave if we performed six iterations. If we were to then go ahead and perform a sixth iteration, many points which still appear to be in the filled Julia set probably don't belong there, and would leave if we performed seven iterations. And so on with each additional iteration. Hence, the number of points in the filled Julia set which don't belong there decreases with each additional iteration. Fifteen iterations is usually sufficient for getting a good sense of the shape of a filled Julia set. Of course, using more than 15 iterations will give a more accurate picture of the filled Julia set. In some situations—particularly for filled Julia sets which are also Cantor sets—the difference may be significant. Consider the filled Julia set of Q_c where $c = -0.75 + 0.1i$ drawn with 15 iterations:

```
> julia_bw(-.75, .1, 20000, 15, 10);
```



Now consider the same filled Julia set when calculated using 100 iterations:

```
> julia_bw(-.75, .1, 20000, 100, 10);
```



The difference is stunning. The filled Julia set which appeared to be a solid connected set in the first image reveals itself to actually be a disconnected set. In fact, if even more than 100 iterations were performed, the few regions which still appear to be solid in this last picture would disappear altogether and give rise to a completely disconnected Cantor set.

I would like to make three observations regarding filled Julia sets.

1. For different values of c, filled Julia sets take on a wide variety of shapes and sizes. Many of the sets have a complicated "organic" look, and the boundaries of these sets are very intricately detailed. Are these strange shapes randomly distributed with c, or is there a way to classify them mathematically? As we will see in the next chapter, filled Julia sets can be nicely classified by the Mandelbrot set.

2. Many filled Julia sets seem to show self-similarity. This would suggest that filled Julia sets are fractals: geometric objects with infinite detail and self-similarity at every level of magnification. To verify this for ourselves, we will need to adjust the `julia_bw` procedure slightly so that we are allowed control over the viewing window of the Julia set. This will allow us to zoom in on small areas of the set:

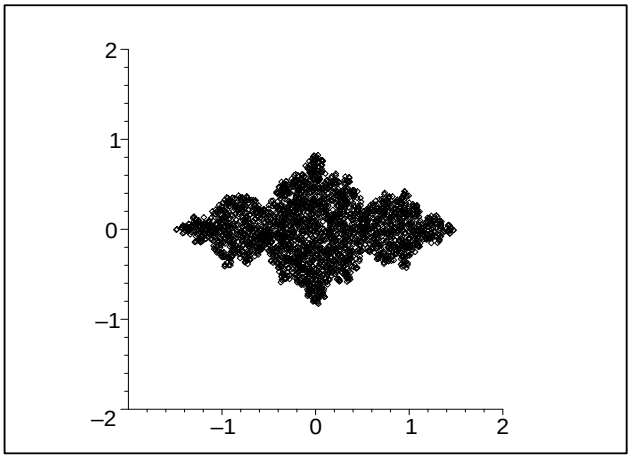
```
> julia_bw_zoom:=proc(a, b, numpoints, maxit, decplaces, xmin, xmax,
> ymin, ymax)
> local X, Y, juliaset, R1, R2, i, j:
> X:=array(0..maxit+1):  Y:=array(0..maxit+1):
> juliaset:=array(0..numpoints):
> juliaset[0]:={}:
> R1:=evalf(rand(trunc(xmin*10^decplaces)..trunc(xmax*10^decplaces))/10
> ^decplaces);
> R2:=evalf(rand(trunc(ymin*10^decplaces)..trunc(ymax*10^decplaces))/10
> ^decplaces);
```

```
> for i from 1 to numpoints do
> X[0]:=R1():  Y[0]:=R2():
> for j from 0 while sqrt(X[j]^2+Y[j]^2)<max(2,sqrt(a^2+b^2)) and
> j<=maxit do
> X[j+1]:=X[j]^2-Y[j]^2+a;
> Y[j+1]:=2*X[j]*Y[j]+b;
> od:
> if j>maxit then juliaset[i]:=(juliaset[i-1] union
> {[X[0],Y[0]]}):
> else juliaset[i]:=juliaset[i-1]:
> end if:
> od:
> with(plots):
> pointplot(juliaset[numpoints], view=[xmin..xmax, ymin..ymax],
> symbolsize=10, symbol=diamond, scaling=constrained, color=black,
> axes=frame):
> end:
```

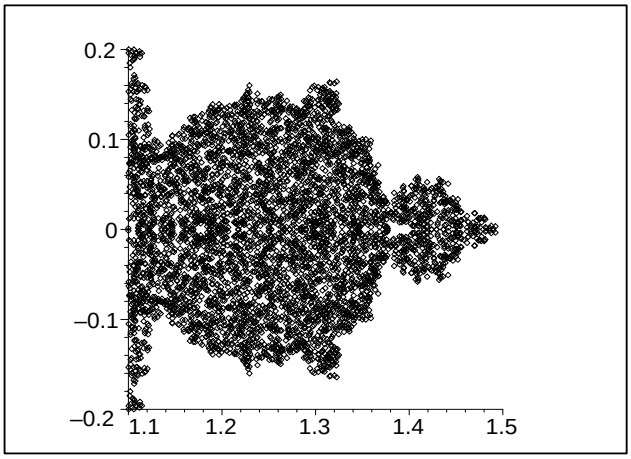
Note that the `julia_bw_zoom` procedure was created by making only a few minor adjustments to the `julia_bw` procedure.

Below we use the `julia_bw_zoom` procedure to plot the filled Julia set for the complex quadratic map with $c = -0.75$ and perform two successive zoom-ins on the right side of the figure:

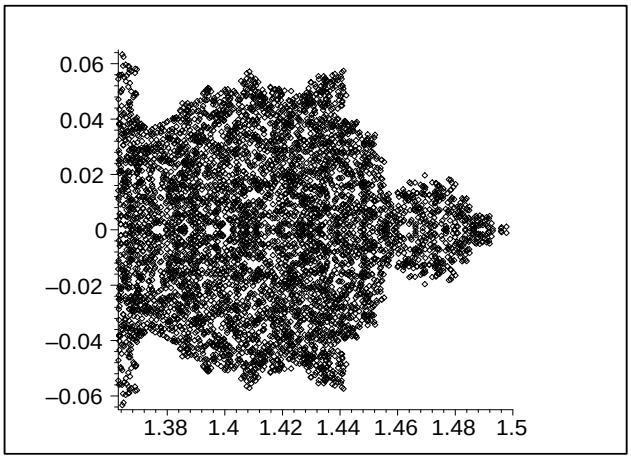
```
> julia_bw_zoom(-.75, 0, 20000, 15, 10, -2, 2, -2, 2);
```




```
> julia_bw_zoom(-.75, 0, 10000, 15, 10, 1.1, 1.5, -.2, .2);
```



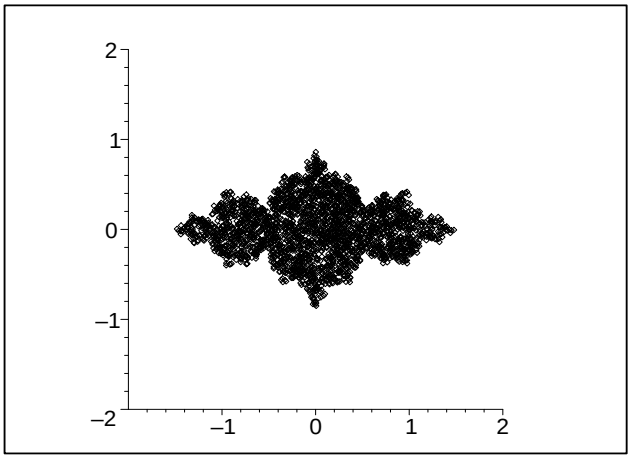

```
> julia_bw_zoom(-.75, 0, 10000, 15, 10, 1.363, 1.5, -.065, .065);
```



This filled Julia set is clearly a fractal.

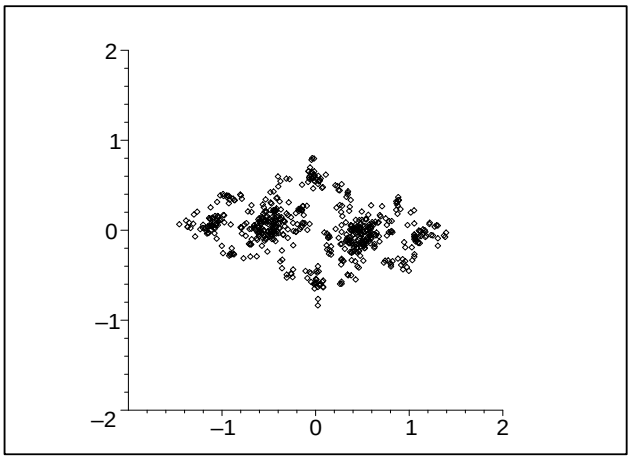
3. And lastly, for some values of c the filled Julia set consists of disconnected isolated points which form a Cantor set. The transition from connected filled Julia sets to Julia sets which are Cantor sets seems to take place suddenly for only small changes in c. The reason for this may be linked to the real quadratic map, $Q_c(x) = x^2 + c$. It is known that for the real quadratic map, a period doubling takes place at $c = -0.75$. Even though we have already plotted the filled Julia set of the complex quadratic map at $c = -0.75$ using the `julia_bw_zoom` procedure above, we will compute it again here using 100 iterations rather than 15. This way we will to be sure we aren't missing any important detail:

```
> julia_bw(-.75, 0, 20000, 100, 10);
```



Compare this filled Julia set with the one that is produced for the complex quadratic map when $c = -0.75 + 0.1i$:

```
> julia_bw(-.75, .1, 20000, 100, 10);
```



Note that when $c = -0.75$, $|c| = 0.75$, and when $c = -0.75 + 0.1i$, $|c| = 0.7566373...$. This small difference in the modulus of c causes the filled Julia set to go from a solid connected set to one that has been "shattered" into a completely disconnected Cantor set. This is similar to the manner in which the orbit diagram of the real quadratic map suddenly goes from a single fixed point orbit to a 2-cycle. But, after all, it shouldn't be all that surprising that a link exists between the real quadratic map and the complex quadratic map.

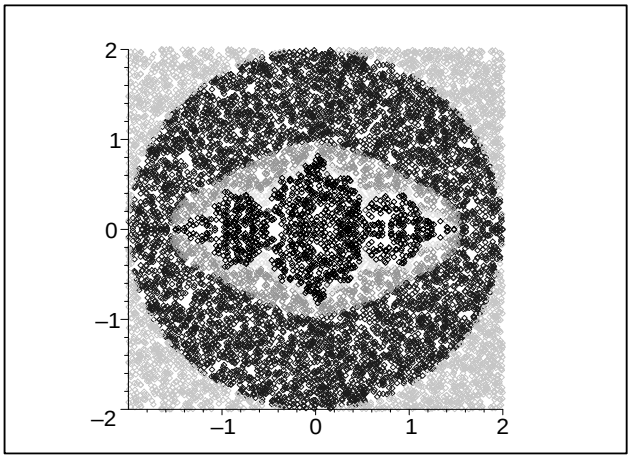
The filled Julia set of a complex function consists only of those points whose orbits are bounded. Very interesting and informative plots can be created, however, if we assign colors to those points which lie outside the filled Julia set according to how quickly their orbits escape to ∞. For instance, we could color all the points which escape most quickly—after, say, one iteration—red. We could color points which leave after 3 or 4 iterations orange. The next set of points to escape would be yellow. Then green. Then blue. Then purple. Various shades of these six main colors as well as mixtures of them can be used for points in between. The number of colors used and the types of colors used are arbitrary. The order of the colors is also arbitrary. Below is a procedure similar to the `julia_bw` and `julia_bw_zoom` which produces filled Julia sets with five different colors:

```
> julia_5color_zoom:=proc(a, b, numpoints, maxit, decplaces, xmin,
> xmax, ymin, ymax)
> local X, Y, juliaset, juliaset1, juliaset2, juliaset3, juliaset4,
> juliaset5, J, J1, J2, J3, J4, J5, R1, R2, i, j:
> X:=array(0..maxit+1):  Y:=array(0..maxit+1):
> juliaset:=array(0..numpoints):
> juliaset1:=array(0..numpoints):
> juliaset2:=array(0..numpoints):
> juliaset3:=array(0..numpoints):
> juliaset4:=array(0..numpoints):
> juliaset5:=array(0..numpoints):
> juliaset[0]:={}:
> juliaset1[0]:={}:  juliaset2[0]:={}:
> juliaset3[0]:={}:juliaset4[0]:={}:  juliaset5[0]:={}:
> R1:=evalf(rand(trunc(xmin*10^decplaces)..trunc(xmax*10^decplaces))/10
> ^decplaces);
> R2:=evalf(rand(trunc(ymin*10^decplaces)..trunc(ymax*10^decplaces))/10
> ^decplaces);
> for i from 1 to numpoints do
> X[0]:=R1():  Y[0]:=R2():
> for j from 0 while sqrt(X[j]^2+Y[j]^2)<max(2,sqrt(a^2+b^2)) and
> j<=maxit do
> X[j+1]:=X[j]^2-Y[j]^2+a;
> Y[j+1]:=2*X[j]*Y[j]+b;
> od:
> if j>maxit then juliaset[i]:=(juliaset[i-1] union {[X[0],Y[0]]}):
> juliaset1[i]:=juliaset1[i-1]:  juliaset2[i]:=juliaset2[i-1]:
> juliaset3[i]:=juliaset3[i-1]:  juliaset4[i]:=juliaset4[i-1]:
> juliaset5[i]:=juliaset5[i-1]:
> else if j>3*trunc(maxit/5) then juliaset[i]:=juliaset[i-1]:
> juliaset1[i]:=(juliaset1[i-1] union {[X[0],Y[0]]}):
> juliaset2[i]:=juliaset2[i-1]:  juliaset3[i]:=juliaset3[i-1]:
> juliaset4[i]:=juliaset4[i-1]:  juliaset5[i]:=juliaset5[i-1]:
```

```
> else if j>1.25*trunc(maxit/5) then juliaset[i]:=juliaset[i-1]:
> juliaset2[i]:=(juliaset2[i-1] union {[X[0],Y[0]]}):
> juliaset1[i]:=juliaset1[i-1]:  juliaset3[i]:=juliaset3[i-1]:
> juliaset4[i]:=juliaset4[i-1]:  juliaset5[i]:=juliaset5[i-1]:
> else if j>trunc(maxit/5)/2 then juliaset[i]:=juliaset[i-1]:
> juliaset3[i]:=(juliaset3[i-1] union {[X[0],Y[0]]}):
> juliaset2[i]:=juliaset2[i-1]:  juliaset1[i]:=juliaset1[i-1]:
> juliaset4[i]:=juliaset4[i-1]:  juliaset5[i]:=juliaset5[i-1]:
> else if j>0 then juliaset[i]:=juliaset[i-1]:
> juliaset4[i]:=(juliaset4[i-1] union {[X[0],Y[0]]}):
> juliaset2[i]:=juliaset2[i-1]:  juliaset3[i]:=juliaset3[i-1]:
> juliaset1[i]:=juliaset1[i-1]:  juliaset5[i]:=juliaset5[i-1]:
> else juliaset[i]:=juliaset[i-1]:  juliaset5[i]:=(juliaset5[i-1]
> union {[X[0],Y[0]]}):  juliaset2[i]:=juliaset2[i-1]:
> juliaset3[i]:=juliaset3[i-1]:  juliaset4[i]:=juliaset4[i-1]:
> juliaset1[i]:=juliaset1[i-1]:
> end if:  end if:  end if:  end if:  end if:
> od:
> with(plots):
> J:=pointplot(juliaset[numpoints], symbolsize=10, symbol=diamond,
> scaling=constrained, color=black):
> J1:=pointplot(juliaset1[numpoints], symbolsize=10, symbol=diamond,
> scaling=constrained, color=red):
> J2:=pointplot(juliaset2[numpoints], symbolsize=10, symbol=diamond,
> scaling=constrained, color=yellow):
> J3:=pointplot(juliaset3[numpoints], symbolsize=10, symbol=diamond,
> scaling=constrained, color=green):
> J4:=pointplot(juliaset4[numpoints], symbolsize=10, symbol=diamond,
> scaling=constrained, color=blue):
> J5:=pointplot(juliaset5[numpoints], symbolsize=10, symbol=diamond,
> scaling=constrained, color=plum):
> display({J,J1,J2,J3,J4,J5}, view=[xmin..xmax, ymin..ymax],
> axes=frame);
> end:
```

Now we will use this procedure to replot the filled Julia set of the complex quadratic map for $c = -0.75$ with colors:

```
> julia_5color_zoom(-.75, 0, 10000, 20, 10, -2, 2, -2, 2);
```



In the `julia_5color_zoom` procedure, we used the same basic method used in the `julia_bw` and `julia_bw_zoom` procedures to compute the filled Julia set. The major difference, and what makes the procedure so lengthy, is addition of five new arrays—named "juliaset1" through "juliaset5"—to hold the ordered (x, y) points which correspond to our five colors. Unfortunately, the nested `if-then` statements, which sort the ordered pairs into these five arrays, cause the algorithm to become painstakingly slow. And for all of the time spent waiting for the calculation to finish, the results are not overwhelmingly aesthetically pleasing.

The method used in this algorithm and those above is the normal way in which we have gone about producing fractals and other plots throughout the course of this manual. It is also the most logical way. What we have done is calculate all the points which belong to the plot, lump them in a big group, and then use `pointplot` to plot them all at once. Because Maple is not able to place pixels on the screen like other programming languages (such as the *putpixel* command in Borland Pascal 7.0), there seems to be no alternative.

Fortunately, this is not the case. A beautiful algorithm has been developed which allows us to use Maple to create filled Julia sets in color with as much intricacy and detail as those we see in books and on posters.[3] Part of this algorithm involves a procedure which I have named `juliafast` (displayed below). The name is appropriate because the algorithm is very, very fast. On my Dell desktop computer with a Pentium III processor, it takes about 20 to 30 seconds to produce a filled Julia set with detail of 250 by 250 pixels using 30 iterations. Upping the number of iterations to 100 (as is necessary to obtain accuracy with some Cantor sets) doesn't noticeably increase this time. How does it work? First, you must understand that Maple has the ability to assign colors to three-dimensional plots according to a two-variable function. This cannot be done for two-dimensional plots. Even though filled Julia sets are two-dimensional sets, we can plot them on a single plane in 3d space

[3]This routine was originally published in the book Maple V: Programming Guide, by M. B. Monagan et al., Springer Verlag.

and take advantage of this capability of Maple. The function we will use to assign the color will actually be the `juliafast` procedure itself. Here is the procedure:

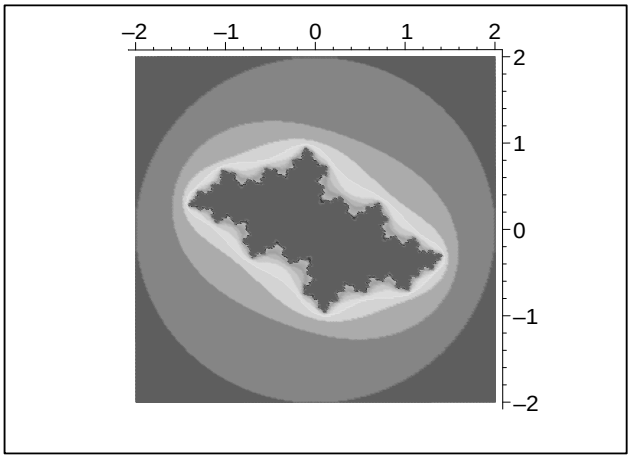
```
> juliafast:=proc(x, y)
> global a, b:
> local x2, y2, x3, i:
> x2:=x;
> y2:=y;
> for i from 0 to maxit while
> evalhf(sqrt(x2^2+y2^2))<max(evalhf(sqrt(a^2+b^2)), 2) do
> x3:=x2;
> x2:=evalhf(x2^2-y2^2+a);
> y2:=evalhf(2*x3*y2+b);
> od;
> i
> end:
```

The nice thing about this procedure as compared with the others used in this chapter is how short it is. We don't need to create arrays because we won't be storing the results of our iteration. All we are interested in is the value of our indexing variable i after the orbit has been calculated. Our coloring function is thus a two-dimensional collection of integers i. To create the filled Julia set, we need to first assign values to a and b and also the name "maxit" used in the procedure to indicate the maximum number of iterations to perform:

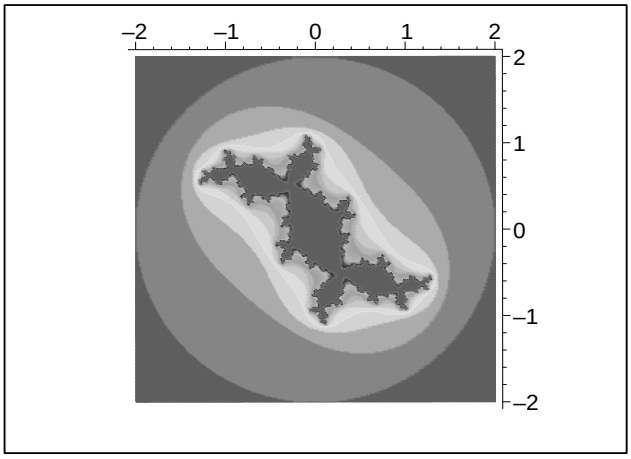
```
> maxit:=20:  a:=-.5:  b:=.5:
```

Next, we use `plot3d` and plot a the plane corresponding to $z = 0$: which is simply the xy-plane. The options used here are also very important in creating a good-looking filled Julia set. Set style=patchnogrid, orientation=[-90,0], and axes=frame. The grid size controls the number of horizontal and vertical pixels to be used to create our plot. Values of 250 and 250 work well. Using fewer pixels will take less time but result in a less detailed plot. Using more pixels will increase the detail of our filled Julia set but take more time. What makes it all work is using the color option and setting it equal to the name of our procedure, `juliafast`. Below I have replotted all of the Julia sets shown in the examples above, but this time using the `juliafast` procedure. The difference is remarkable:

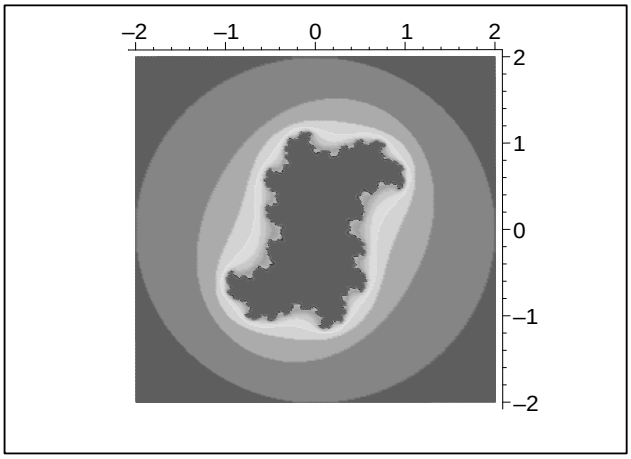
```
> plot3d(0, -max(sqrt(a^2+b^2),2)..max(sqrt(a^2+b^2),2),
> -max(sqrt(a^2+b^2),2)..max(sqrt(a^2+b^2),2), style=patchnogrid,
> orientation=[-90,0], grid=[250,250], scaling=constrained, axes=frame,
> color=juliafast);
```



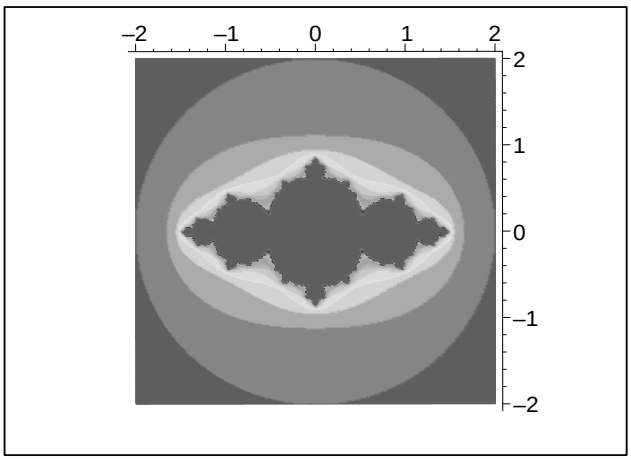

```
> a:=-.1:  b:=.8:
> plot3d(0, -max(sqrt(a^2+b^2),2)..max(sqrt(a^2+b^2),2),
> -max(sqrt(a^2+b^2),2)..max(sqrt(a^2+b^2),2), style=patchnogrid,
> orientation=[-90,0], grid=[250,250], scaling=constrained, axes=frame,
> color=juliafast);
```




```
> a:=.3:  b:=-.4:
> plot3d(0, -max(sqrt(a^2+b^2),2)..max(sqrt(a^2+b^2),2),
> -max(sqrt(a^2+b^2),2)..max(sqrt(a^2+b^2),2), style=patchnogrid,
> orientation=[-90,0], grid=[250,250], scaling=constrained, axes=frame,
> color=juliafast);
```

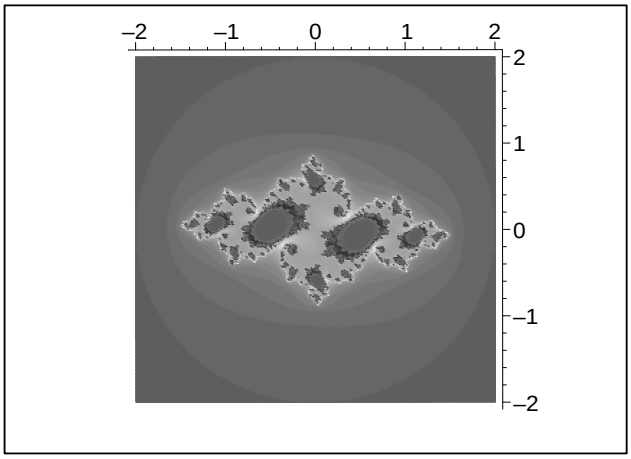



```
> a:=-.75:  b:=0:
> plot3d(0, -max(sqrt(a^2+b^2),2)..max(sqrt(a^2+b^2),2),
> -max(sqrt(a^2+b^2),2)..max(sqrt(a^2+b^2),2), style=patchnogrid,
> orientation=[-90,0], grid=[250,250], scaling=constrained, axes=frame,
> color=juliafast);
```



To get good detail on the Cantor set which is produced for $c = -0.75 + 0.1i$ we set maxit=100:

```
> maxit:=100:  a:=-.75:  b:=.1:
> plot3d(0, -max(sqrt(a^2+b^2),2)..max(sqrt(a^2+b^2),2),
> -max(sqrt(a^2+b^2),2)..max(sqrt(a^2+b^2),2), style=patchnogrid,
> orientation=[-90,0], grid=[250,250], scaling=constrained, axes=frame,
> color=juliafast);
```



It is easy to zoom in on these plots, if you desire, simply by adjusting the ranges in `plot3d` to the exact values that you want. The one unfortunate drawback of the `juliafast` procedure is that you have no control over the colors used to produce the filled Julia set. The colors are automatically chosen by Maple. Usually, `juliafast` uses red to color points whose orbits are bounded and which actually belong to the filled Julia set, rather than using the traditional black (as in our black and white plots from `julia_bw`).

10.1 Experiment: Filled Julia Sets for Q_c

1. Plot 10 or more different black and white filled Julia sets for the complex quadratic map, Q_c. Do not re-plot any of the filled Julia sets which are shown as examples earlier in this chapter. Try to plot filled Julia sets from a diverse range of values of a and b. The most interesting filled Julia sets occur for $\sqrt{a^2+b^2} \lessapprox 2$. Be aware, however, that many values of a and b do not produce nice filled Julia sets. For some values you will see only a few scattered points. This can be true even when $\sqrt{a^2+b^2} \lessapprox 2$. Therefore, you will have to use trial and error to find 10 or more interesting plots. (If you are familiar with the Mandelbrot set—which will be discussed in Chapter 11 below—then you will not need to use trial and error. Simply choose values of (a,b) which belong to the Mandelbrot set or lie near its border.)

2. Choose one of your filled Julia sets from the step above and perform two or more successive zoom-ins to show the fractal nature of filled Julia sets. When you zoom-in you will most likely need to increase the number of iterations performed ("maxit") in order to maintain detail. Also, depending on the region you wish to zoom-in on, you may need to set scaling=unconstrained so that the plot can be stretched to a comfortable viewing size.

3. Re-plot all of the filled Julia sets which you plotted in step 1 above using the `juliafast` procedure from earlier in this chapter. These filled Julia sets will now appear in color, with the color indicating the speed at which points that are not in the filled Julia set

escape to ∞. Notice how much faster the `juliafast` procedure is at creating filled Julia sets and also the increased amount of detail in the filled Julia sets. For some of your filled Julia sets you may want to increase your maximum number of iterations to increase the detail and accuracy of your filled Julia set. (For instance, connected filled Julia sets become clearly formed after only 10 or 20 iterations, but filled Julia sets which are Cantor sets may require 100 iterations to become accurately formed). In most cases, this will not significantly increase the time it takes to plot the filled Julia set.

4. Choose on of your color filled Julia sets from the step above and perform two or more successive zoom-ins to show the fractal nature of the the set. When you zoom-in you will most likely need to increase the number of iterations performed ("maxit") in order to maintain detail. Also set scaling=unconstrained so that the graph can be stretched to a comfortable viewing size.

5. If time permits, use `juliafast` to create several more new filled Julia sets for Q_c. Also try zooming-in on portions of these sets. This should be an enjoyable pursuit because of how quickly `juliafast` is able to plot the sets.

10.2 Experiment: Filled Julia Sets for Other Functions

1. $E_c(z) = ce^z$

   (a) Write a fast algorithm for producing filled Julia sets of the family of complex functions $E_c(z) = ce^z$, where $c = a + ib$ and $z = x + iy$. This can be done by making adjustments to the iterative equations in the `juliafast` algorithm which was used earlier in this chapter to create filled Julia sets of the family of complex quadratic functions, Q_c. You will also have to use a different escape criterion:

   Theorem 10.2 (Approximate Escape Criterion for Exponential Functions) *If* $|\Re(E_c^n(z))| > 50$*, then* $E_c^n(z) \to \pm\infty$ *as* $n \to \infty$.

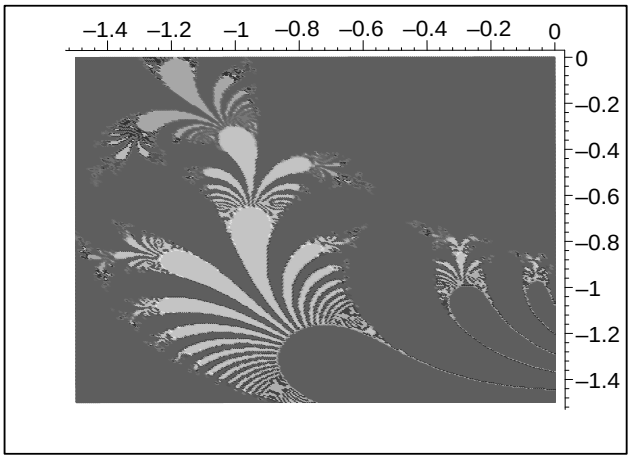
   HINT: Using Euler's formula, $e^{i\alpha} = \cos\alpha + i\sin\alpha$, the real part of $E_c(z)$, $\Re(E_c(z))$, can be found as follows:

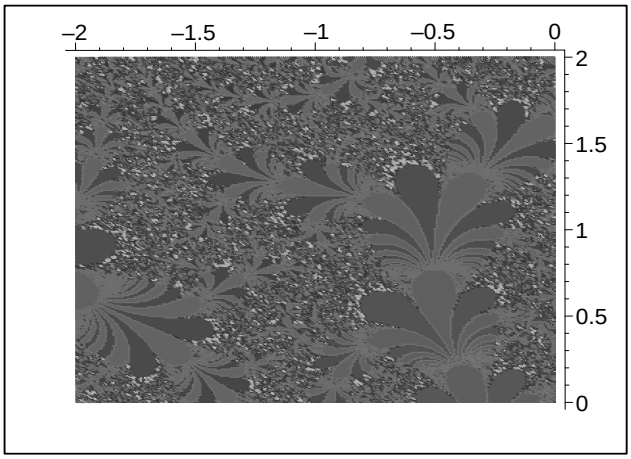
$$\begin{aligned} E_c(z) &= ce^z \\ &= (a+ib)e^{x+iy} \\ &= (a+ib)e^x e^{iy} \\ &= (ae^x + ibe^x)(\cos y + i\sin y) \\ &= (ae^x\cos y - be^x\sin y) + i(be^x\cos y + ae^x\sin y) \end{aligned} \tag{10.9}$$

   Therefore, the real part of $E_c(z)$ is $\Re(E_c(z)) = (ae^x\cos y - be^x\sin y)$. Be alert that this escape criterion is not as concrete as the escape criterion for $Q_c(z)$. It is not exactly true that every point which reaches the half-plane $|\Re(z)| > 50$ necessarily escapes. However, it can be shown that there is usually a point

arbitrarily close which does in fact escape. This subtle point is not important: for our purposes, Theorem 10.2 is adequate.[4]

(b) Plot 10 or more filled Julia sets for the complex family of functions ce^z. Try to plot filled Julia sets from a diverse and interesting range of values of a and b. Portions of a couple of these sets are pictured below. The first corresponds to $a = 1$ and $b = 2$. The second is for $a = 0$ and $b = 2\pi$.





(c) Choose one of your filled Julia sets from step (b) above and perform two successive zoom-ins on the set to show its fractal nature. Don't forget that you will need to increase the number of iterations ("maxit") if you want to maintain intricacy and detail. Also, you may want to set scaling=unconstrained.

2. $S_c(z) = c \sin z$

[4]For more details see Durkin, Marilyn B., "The Accuracy of Computer Algorithms in Dynamical Systems," International Journal of Bifurcation and Chaos. 1 No. 3, 1991.

(a) Write a fast algorithm for producing filled Julia sets of the family of complex functions $S_c(z) = c \sin z$, where $c = a + ib$ and $z = x + iy$. This can be done by making adjustments to the iterative equations in the `juliafast` algorithm which was used earlier in this chapter to create filled Julia sets of the family of complex quadratic functions, Q_c. Use the following escape criterion:

Theorem 10.3 (Approximate Escape Criterion for Trigonometric Functions) *If* $|\Im(S_c^n(z))| > 50$*, then* $S_c^n(z) \to \pm\infty$ *as* $n \to \infty$*.*

Here $\Im(S_c^n(z))$ symbolizes the imaginary part of the function $S_c(z)$. As in the case of the Approximate Escape Criterion for Exponential functions, Theorem 10.3 is not concrete. Not every point whose orbit satisfies $|\Im(S_c^N(z))| > 50$ for some N will necessarily escape to infinity. Arbitrarily close points will, however. *HINT*: use the formula $\sin\alpha = \frac{e^{i\alpha} - e^{-i\alpha}}{2i}$, which follows directly from Euler's formula.

(b) Plot 10 or more filled Julia sets for the complex family of functions $c \sin z$. Try to plot filled Julia sets from a diverse and interesting range of values of a and b.

(c) Choose one of your filled Julia sets from step (b) above and perform two successive zoom-ins on the set to show its fractal nature. Don't forget that you will need to increase the number of iterations ("maxit") if you want to maintain intricacy and detail. Also, you may want to set scaling=unconstrained.

3. $C_c(z) = c \cos z$

(a) Write a fast algorithm for producing filled Julia sets of the family of complex functions $C_c(z) = c \cos z$, where $c = a + ib$ and $z = x + iy$. This can be done by making adjustments to the iterative equations in the `juliafast` algorithm which was used earlier in this chapter to create Julia sets of the family of complex quadratic functions, Q_c. Use the the escape criterion given by Theorem 10.3. *HINT*: Use the formula $\cos\alpha = \frac{e^{i\alpha} + e^{-i\alpha}}{2}$, which follows directly from Euler's formula.

(b) Plot 10 or more filled Julia sets for the complex family of functions $c \cos z$. Try to plot filled Julia sets from a diverse and interesting range of values of a and b.

(c) Choose one of your filled Julia sets from step (b) above and perform two successive zoom-ins on the set to show its fractal nature. Don't forget that you will need to increase the number of iterations ("maxit") if you want to maintain intricacy and detail. Also, you may want to set scaling=unconstrained.