Displaying the Internal Structure of the Mandelbrot Set

Adam Cunningham
State University of New York at Buffalo
Numerical Analysis MTH 537
adamcunn@buffalo.edu

December 20, 2013

Abstract

Although images of the Mandelbrot set are ubiquitous, the detail shown in the majority of such images is in fact of the complement of the set, while the set itself is usually displayed in black. We consider here the problem of the display of the internal structure of the Mandelbrot set itself. Seven different methods for displaying the internal structure are developed, based on considerations of the geometry of the orbit associated with each point in the set. The resulting images obtained using these methods are presented and analyzed.

1 Introduction

Since the work of Benoit Mandelbrot (1924 - 2010) first came to widespread public attention in the 1980s[1], images of the "Mandelbrot Set" have become ubiquitous. These images, both complex and beautiful, introduced a wider audience to the areas of fractal geometry and to the deep connections between this area of mathematics and self-similar structures present in the natural world. The detail present in most of these images is however for points not in the set, but rather in its complement. Points in the set are usually colored black, and show no internal features.

The problem addressed by this report is that of displaying the internal structure of the Mandelbrot set. Several methods for doing so are developed and presented, either variations of existing techniques or (to the authors best knowledge), new techniques for displaying this structure.

In section 2 following we first give a definition of the Mandelbrot set and describe its main components. We then describe the most common algorithm used to create images of the set, the "escape-time algorithm", and discuss some issues that arise with this algorithm when images of the set are magnified.

Section 3 on "The Internal Structure of the Mandelbrot Set" constitutes the main part of this report. Several different methods for displaying the internal structure are described and images generated using these methods are presented and discussed.

In section 4 we compare some of the results obtained with previously published images, and discuss their similarities and differences.

In the final section 5, we analyze roundoff errors involved in computing images of the set and the degree to which the images presented are accurate under magnification.

2 The Mandelbrot Set

We first consider the definition of the Mandelbrot set and the primary way in which the set is usually displayed using the 'escape-time' algorithm.

2.1 Definition of the Mandelbrot Set

Consider the function P_c of a complex variable z defined for a parameter $c \in \mathbb{C}$ by

$$P_c(z) = z^2 + c$$

Starting with the initial value $z_0 = 0$, a sequence of complex values $0, c, c^2 + c, \ldots$ can be defined by

$$\begin{aligned} z_0 &= 0 \\ z_{n+1} &= P_c(z_n) = z_n^2 + c \end{aligned} \tag{1}$$

We shall denote the nth element of the sequence generated using the complex parameter c by P_c^n. It can be readily seen that the behavior of this sequence is dependent on the particular value chosen for c, as the following examples demonstrate:

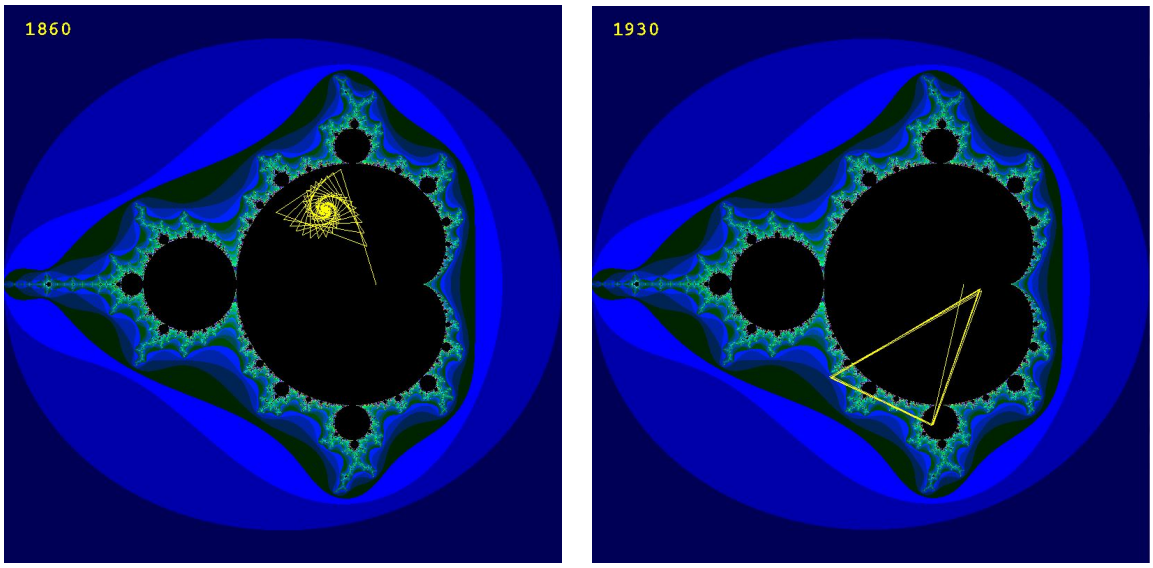
$$\begin{array}{rcrcrcrcrcrcl} c & = & 0: & 0 & \to & 0 & \to & 0 & \to & 0 & \to & 0 & \ldots \\ c & = & -0.5: & 0 & \to & -0.5 & \to & -0.25 & \to & -0.4375 & \to & -0.3086 & \to & -.4048 & \ldots \\ c & = & -1: & 0 & \to & -1 & \to & 0 & \to & -1 & \to & 0 & \to & -1 & \ldots \\ c & = & 1: & 0 & \to & 1 & \to & 2 & \to & 5 & \to & 26 & \to & 677 & \ldots \end{array}$$

Depending on the value of c, the sequences shown above either converge (to a single point or periodic cycle), or else diverge to infinity. In the first case, the sequence is bounded, while in the second case, the sequence is unbounded. Denoting the Mandelbrot set by $\mathscr{M}$, a formal definition is given by

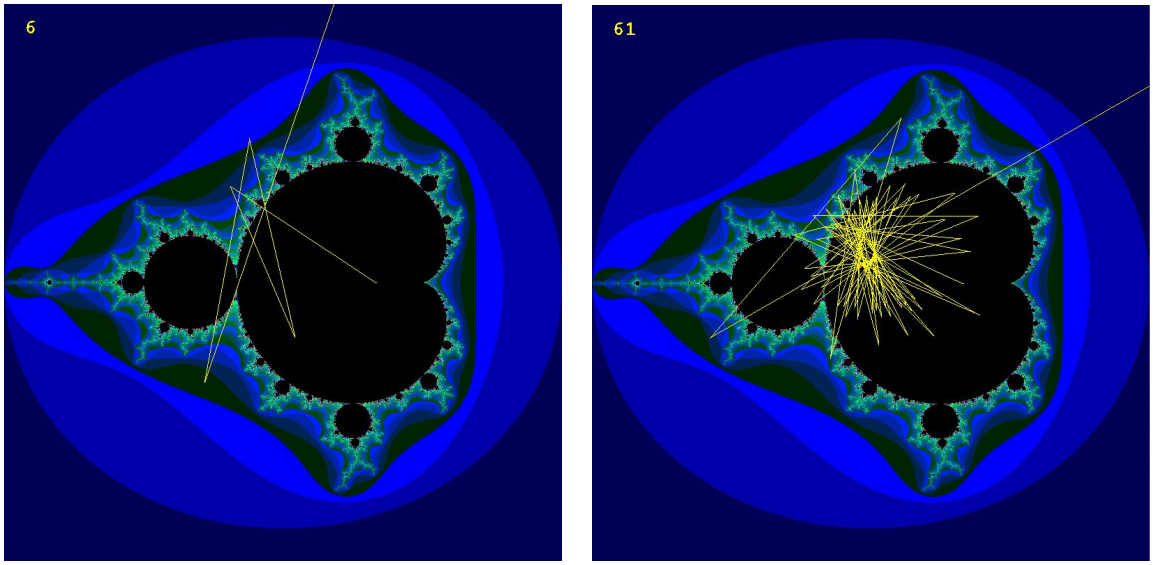
$$\mathscr{M} = \{c \in \mathbb{C} \mid P_c^n \not\to \infty\}$$

The Mandelbrot set is therefore defined as the set of all values of the parameter c in the complex plane $\mathbb{C}$ for which the sequence (1) remains bounded. From the examples above it can be seen that the points 0, -1/2 and -1 are in the set, while the point 1 is not.

The following figures illustrate the two kinds of sequences, which shall be referred to as **orbits**. The diagrams will be referred to as **orbit diagrams**, which plot successive points in an orbit as points in the complex plane connected by straight lines. The first two figures show orbits which do not diverge, and which therefore correspond to points $c \in \mathscr{M}$. The orbit on the left converges to a single point, while that on the right converges to a **cycle** of period 3, a sequence of three points that repeats infinitely.



The figures below illustrate two orbits which *do* diverge, and which therefore correspond to parameters $c \notin \mathscr{M}$. In both cases $|P_c^n| > 2$ for some positive integer n. The value of n at which this occurs is shown in the upper left of each figure.



2.2 Geometry of the Mandelbrot Set

It is useful to give a brief description of the geometry of the set.

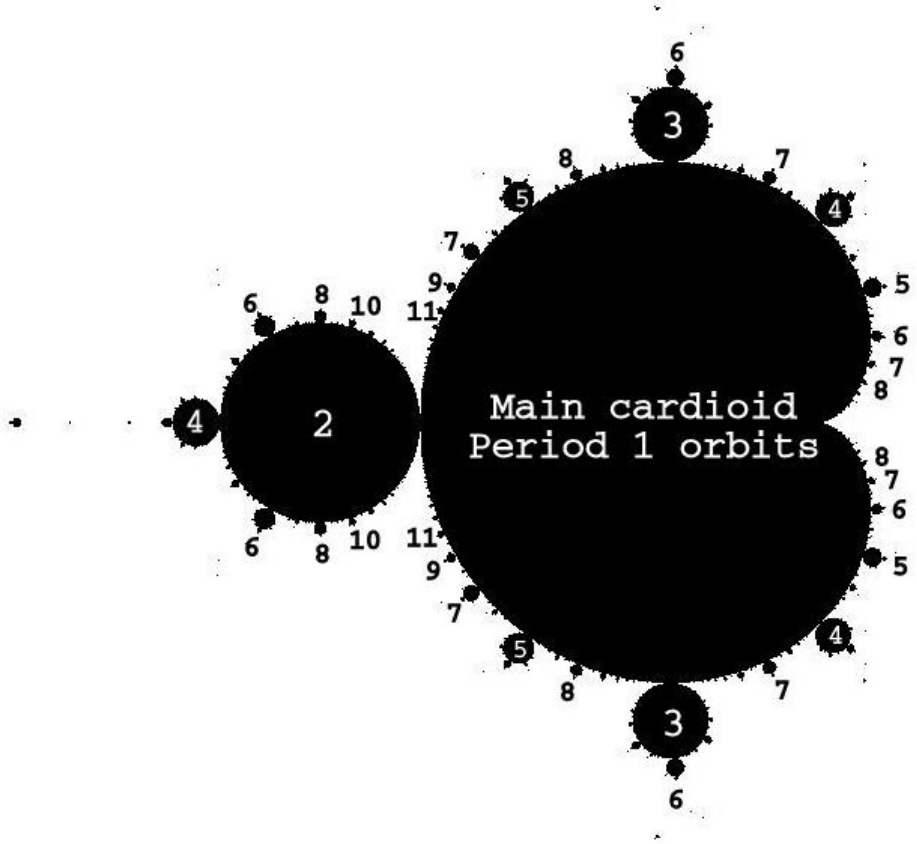
1. The large central component is the **main cardioid**, which contains the parameters c for which P_c^n converges to a single point.
2. Tangent to the main cardioid are an infinite number of **period bulbs**, each one corresponding to a set of parameters giving rise to orbits of the same period. The largest of these is shown to

the left of the main cardioid and contains parameters c which give rise to orbits that converge to cycles of period 2. The next largest bulbs directly above and below the main cardioid contain parameters c which give rise to orbits that converge to cycles of period 3.

3. Surrounding the set, although connected to it by thin structures that are in the set, are an infinite number of **satellites**, smaller copies of the main body of the set.

The ratio between the period of the attracting cycle of the main cardioid and the tangent period bulbs is reproduced between these period bulbs and the bulbs which are tangent to them in turn. So, the bulbs tangent to the main period 2 bulb have periods that are multiples of 2, the bulbs tangent to the main period 3 bulbs have periods that are multiple of 3, and this same pattern is repeated for period bulbs of longer period.

The following diagram labels these components, which we shall refer to throughout this report. The numbers indicate the period of the cycle that orbits converge to in the closest period bulb.



2.3 Displaying the Mandelbrot Set

The images of the Mandelbrot set shown above are constructed by dividing the area to be displayed into pixels, taking the value c to be the center of the pixel, and applying the iteration formula (1). The simplest way to display the set is by coloring points $c \in \mathbb{C}$ black if they are in the set and white otherwise. A more sophisticated display method use different colors to indicate the number of iterations n needed for $|P_c^n|$ to exceed some threshhold, as described below for the "escape-time algorithm".

2.3.1 The Escape Time Algorithm

It can be shown that if the absolute value attained by a point z_n in an orbit is ever larger than both $|c|$ and 2, then the sequence P_c^n will diverge to infinity. Suppose that both $|z_n| > 2$ and $|z_n| > c$. Then $|z| \geq 2 + \epsilon$ for some positive ϵ, and we therefore have:

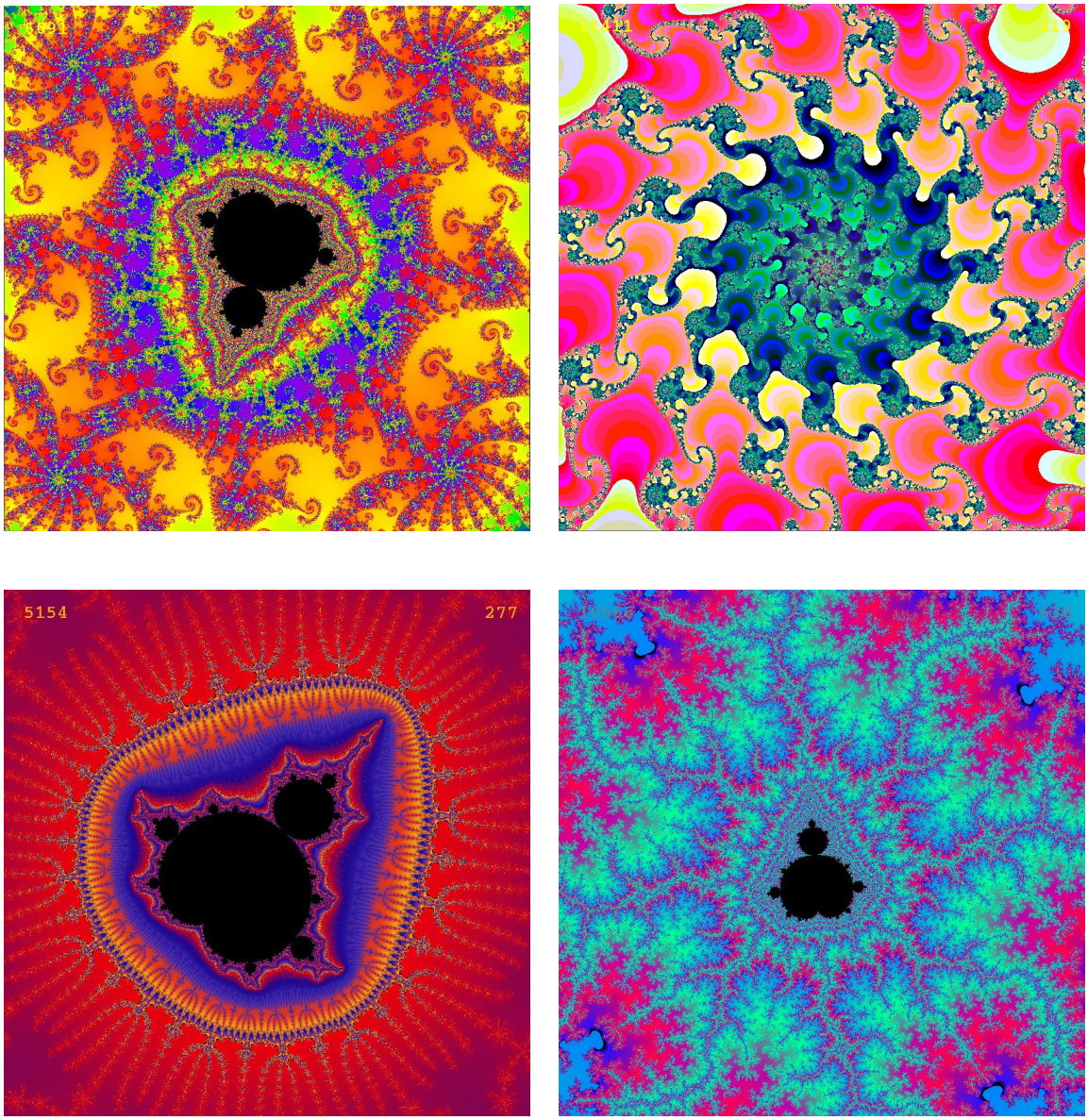
$$\begin{aligned} |z_{n+1}| &= |z_n^2 + c| \\ &\geq |z_n^2| - |c| \\ &\geq |z_n^2| - |z| \\ &= |z|(|z| - 1) \\ &\geq |z|(1 + \epsilon) \end{aligned}$$

Repeating this process yields $|z_{n+k}| \geq |z|(1+\epsilon)^k \to \infty$. The "escape time algorithm" described below uses this test of divergence to color pixels according to how many iterations n are needed until $|z_n| > 2$.

Algorithm 1 Escape-Time Algorithm

Require: $C \in \mathbb{C}$
  C = grid of points in a region of the complex plane
  Z = grid of zero complex values
  Image = grid of zero integers
  iteration = 0
  loop
    $Z = Z^2 + C$
    iteration = iteration + 1
    Image[$|Z| > 2$] = iteration
    display Image
  end loop

The following images were created using the escape-time algorithm and different color maps.



We note that:

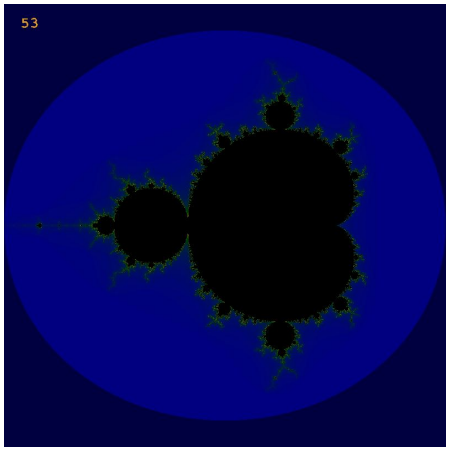
1. For points in the set, the threshhold $|z_n| > 2$ is never reached. In the images of the Mandelbrot set most commonly seen these points are usually colored black.
2. For points far from the set (and immediately for points outside the circle $|z| = 2$), this value is reached in just a few iterations.
3. For points outside of but close to the set, the number of iterations needed until $|z_n| > 2$ increases without bound as the distance from the set decreases.

2.4 Magnification of the Set

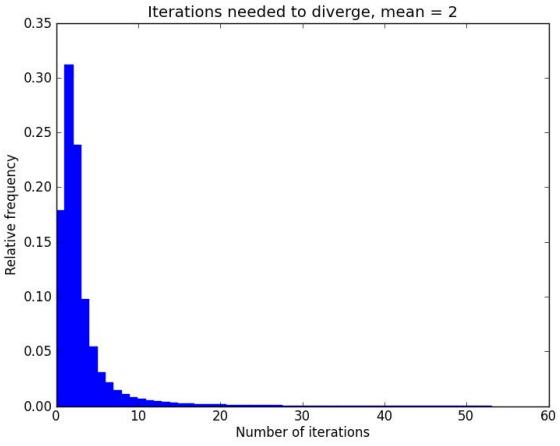
When an image of the Mandelbrot set is magnified, the magnified region gives rise to a new set of points in the complex plane, corresponding to the centers of the pixels that the region is partitioned into for display. The simplest way to magnify the set is to compute the color for each new pixel using the escape-time algorithm. This however is computationally expensive, so ways to reduce this are needed.

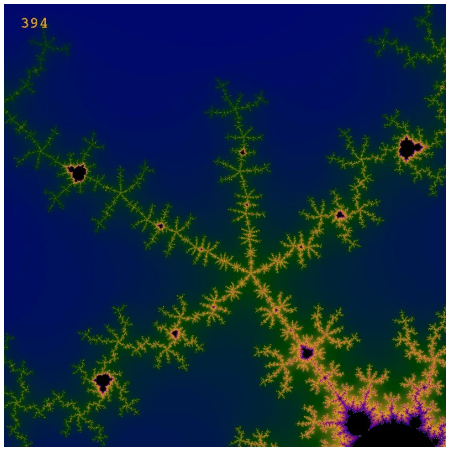
The central problem that arises in magnifying the Mandelbrot set is that the number of iterations needed to diverge is (on average) greater for points which are closer to the set. When an area of the complex plane is magnified, typically this involves magnifying a region close to the edge of the set (as this constitutes the "interesting" part of the image).

This can be seen in the sequence of images of increasing magnification shown below. The images on the left shows regions in the complex plane, magnified by a factor of approximately 25 fold each time relative to image above. The histograms on the right shows the distribution of the number of iterations needed to diverge for the points shown in the images on the left. The mean number of iterations needed is also shown.

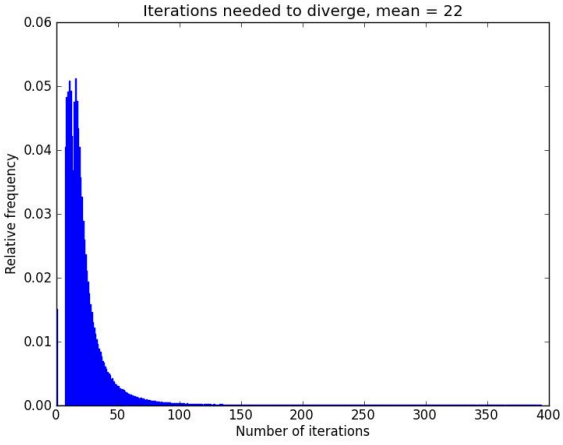


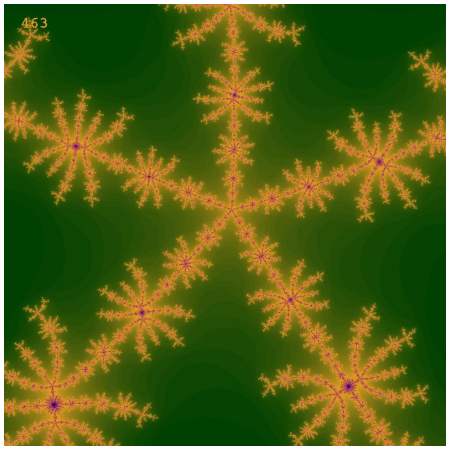
Width = 3.0

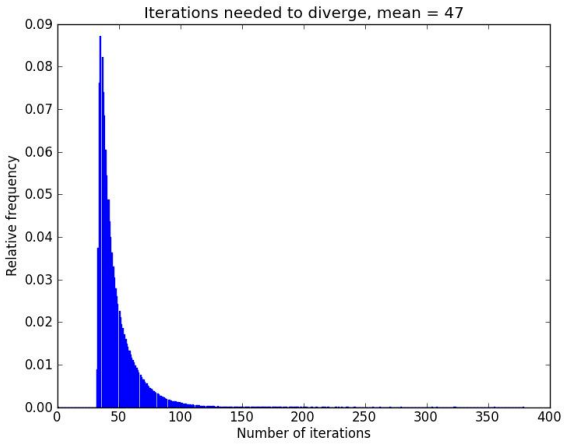




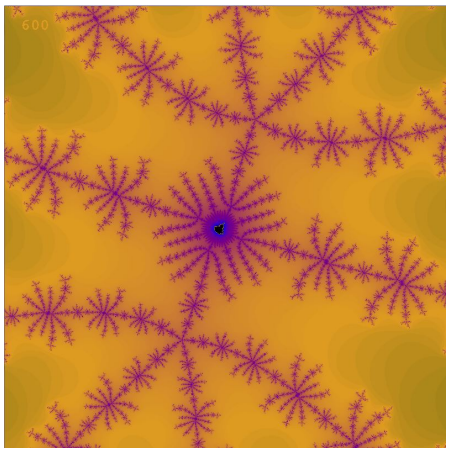
Width = 0.09

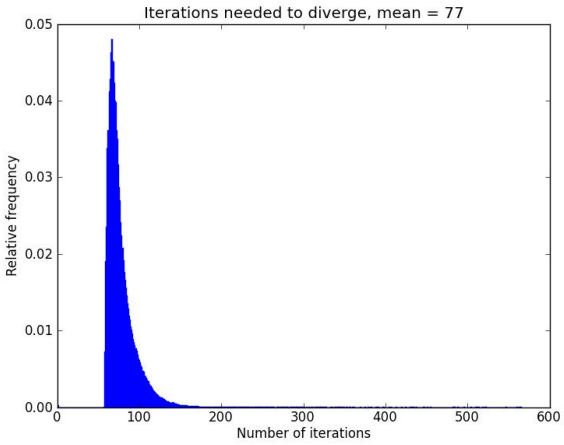






Width = 0.0046





Width = 0.0002

The following observations can be made:

1. The mean number of iterations needed is greater with increasing magnification, resulting in longer times for the image to display.
2. The histograms are "right-shifted" with increasing magnification, as the minimum number of iterations needed to display *any* point in the image increases. When displaying an image, this results in an increasing delay before any part of the image is shown, as a greater number of iterations are needed before the magnitude of any orbit becomes greater than 2.

The issue of magnifying the Mandelbrot set is only pursued further in this report in the context of a discussion of roundoff error in section 5. It can however be noted that magnification can be conceptualized as a form of interpolation between points for which the number of iterations needed is already known. Speeding up the magnification of a region of the set using previously calculated results would therefore require an analysis of how values calculated using the escape-time algorithm could be interpolated to a new set of points.

3 The Internal Structure of the Mandelbrot Set

The most frequently seen images of the Mandelbrot set show points in the set in black, while the escape-time algorithm colors points in the sets complement. The focus of this report is on ways of displaying the internal structure of the set, which has received much less attention.

The question naturally arises as to which kinds of information to display for points inside the set. For points outside the set, the number of iterations needed to diverge, as displayed by the escape-time algorithm, defines a function mapping points in the complex plane to integers, which can then be displayed as colors. For points *inside* the set, a natural counterpart to this function would be one that maps points in the set to the number of iterations needed to *converge* to a set tolerance, either converging to a point or to a periodic cycle. Such a method is developed later in this report and the results presented.

Examination of these results indicates that this approach loses much information of interest. From the orbit diagrams shown earlier, it can be seen that one of the primary difference between points in the set is found in the structure of the orbits that they give rise to. Orbits differ not only in the period of the cycle to which they converge, but also in the rate of convergence and in the period of the orbit as it converges to this cycle. It can also be seen visually that different orbits have geometries that differ in interesting ways. An approach to the internal structure of the set that focuses on the geometry of the orbits therefore has the potential to produce some interesting results.

The previous section showed several images of orbits, where successive points in an orbit were plotted and connected by straight lines. It can be seen that orbits differ in their geometry, with orbits of period greater than one in the shape of simple polygons both convex and concave, or in the shape of star polygons. Orbits that converge to a point display similarly complex structure, with orbits that look like simple convex and star polygons, decreasing in size with increasing number of iterations.

A fundamental issue in displaying the internal structure of the Mandelbrot set in two-dimensions is to find ways to collapse the complex geometric shape of an orbit to a single real number. If this can be done, then the number can be mapped to a color, and an image produced for the set showing the color associated with each point. The possibility also exists of displaying two pieces of information about a point in a single color, where color hue and luminescence are used to plot two values independently.

The following sections describe seven methods developed for this report to display information about the orbits. The first two methods display graphs pertaining to a single point or to a restricted set of such points. The last five methods show images of the whole set, with each displayed point c color-coded to show particular details of the associated orbit P_c^n. These seven methods are:

Orbit Magnitude Plot Plot the distance $|P_c^n - c|$ against n for a single point c.

Feigenbaum Diagrams Plot the distances $|P_c^n - c|$ between points in an orbit against $|c|$ for a restricted set of points $c \in \mathbb{C}$ once the orbit has converged to a periodic cycle.

Orbit Convergence Display for a region the number of iterations needed for the orbit to converge to a periodic cycle to within a given tolerance.

Orbit Cycle Display for a region plane the period of the cycle that the orbit converges to.

Orbit Period Display for a region plane the index n for which $|P_c^n - c|$ is a minimum.

Orbit Infimum Display for a region the minimum value of $|P_c^n - c|$.

Total Orbit Angle Display for a region the sum of the internal angles of the first period of the orbit P_c^n.

3.1 Orbit Magnitude Plot

As a first step towards displaying the internal structure of the set, we developed a method to display a plot of the distance $|P_c^n - c|$ between successive points in an orbit P_c^n and the parameter c. These distances are then plotted against the iteration number n.

The motivation for this method is to visualize the way in which orbits converge to periodic cycles. It will be shown that this distance provides information about both the period of an orbit and the rate of convergence to a periodic cycle. In the following sections methods for displaying images of the internal structure of the Mandelbrot set based on these two parameters will be presented.

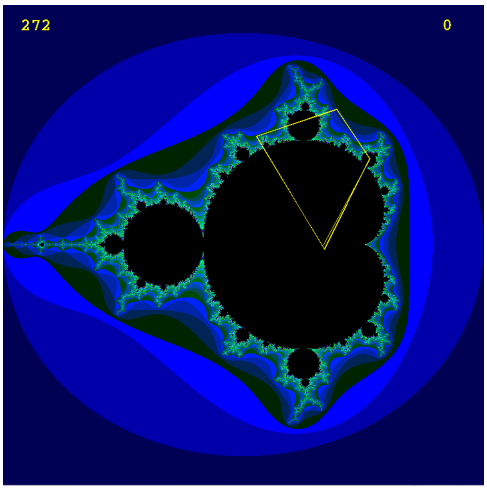
Algorithm 2 Orbit Magnitude Plot

Require: $c \in \mathbb{C}$
  $z = 0$
  iterations = 30
  distance[0] = 0
  for $i = 1$ **to** iterations **do**
    $z = z^2 + c$
    distance[i] = $\|z - c\|$
  end for
  plot distance

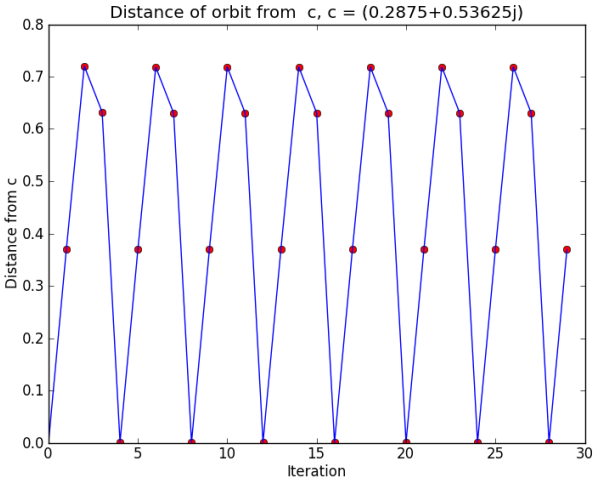
The images and graphs in the following sections show the orbits and orbit magnitude plots for several points c in the complex plane. The image on the right shows the orbit P_c^n. The graph on the left shows the distance $|P_c^n - c|$ plotted against n.

3.1.1 Stable Period 4 Orbit

The orbit shown below was generated for a point in the largest period 4 bulb tangent to the upper right of the main cardioid. On the left, the orbit can be seen to converge to the shape of a convex quadrilateral. On the right, the orbit magnitude plot shows a stable cycle of period 4. So, from the orbit magnitude plot - a graph - we can find information about the shape of the orbit.



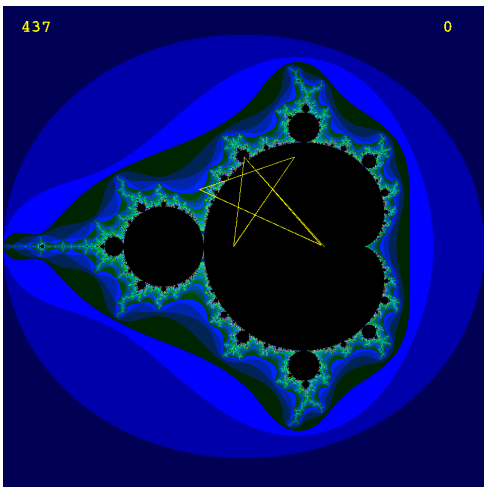
Orbit converges to convex four-sided polygon



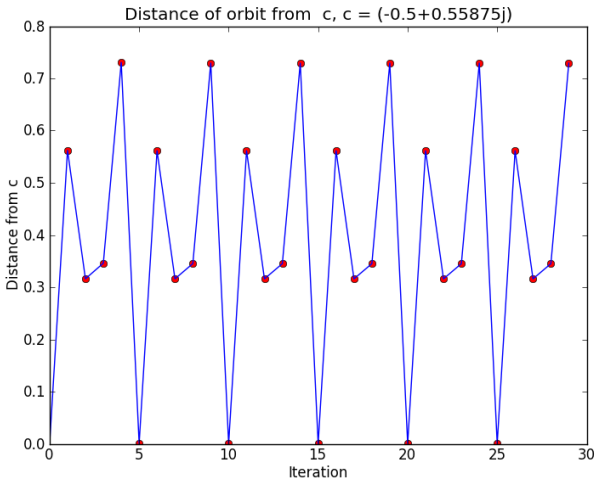
Stable period 4 attractor

3.1.2 Stable Period 5 Orbit

The next orbit shown below was generated for a point in the period 5 bulb tangent to the upper left of the main cardioid. On the left, the orbit can be seen to converge to a five-pointed star polygon. On the right, the orbit magnitude plot shows a stable cycle of period 5, corresponding to the shape of the orbit.



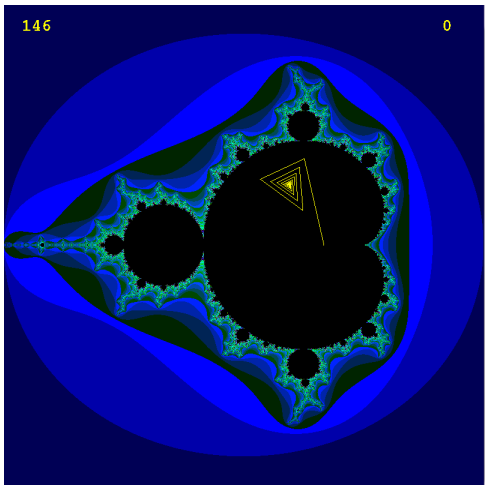
Orbit converges to five-pointed star polygon



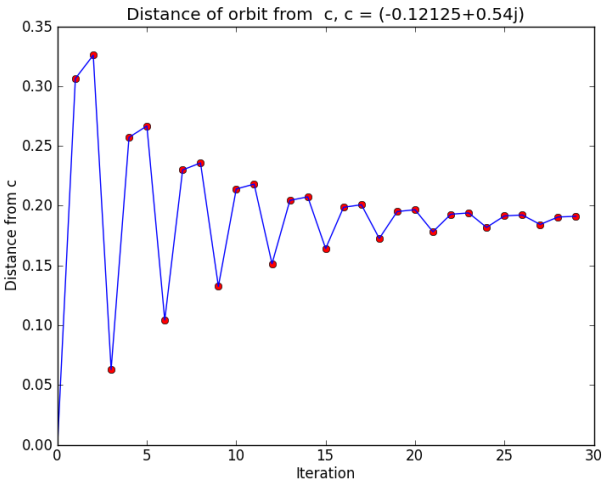
Stable period 5 attractor

3.1.3 Decaying Period 3 Orbit

The orbit shown below was generated for a point inside the main cardioid, but close to the largest upper period 3 bulb. The orbit on the left can be seen to be a "shrinking triangle" that converges to a point. The orbit magnitude plot on the right shows this as a cycle of period 3 bounded by an exponentially decaying envelope.



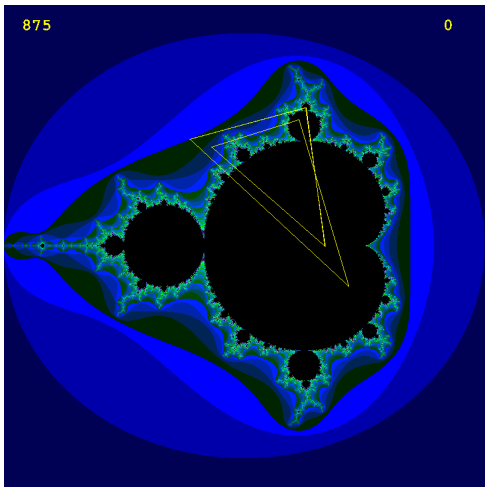
Orbit converges to a point



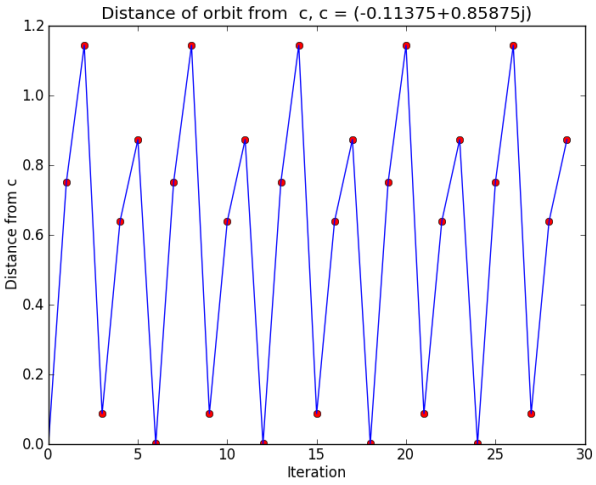
Decaying period 3 attractor

3.1.4 Stable Period 6 Orbit

The orbit shown below was generated for a point in the largest secondary period bulb tangent to the largest upper period 3 bulb. The orbit on the left can be seen to be a "double triangle" attractor of period 6. The orbit magnitude plot on the right shows this as a cycle of period 6..



Orbit converges to period 6 attractor



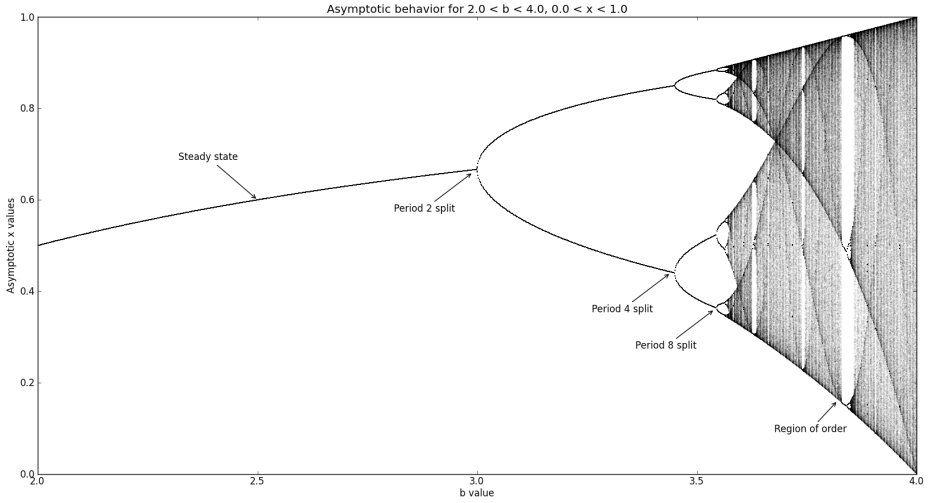
Stable period 6 attractor

In the next section we turn to ways in which information about orbits can be displayed for sets of points, rather than just for single points.

3.2 Feigenbaum Diagrams

The one-dimensional discrete logistic map is generated by the recurrence relation $x_{n+1} = bx_n(1-x_n)$. As with the Mandelbrot set, the behavior of sequences generated by this recurrence relation depend on a single parameter, in this case the real parameter b. For different values of b, the orbits generated may converge to a single point, converge to a periodic attractor, or may wander chaotically.

A frequently used method to illustrate the dependence of the orbits on the parameter b is the "Feigenbaum diagram" or "bifurcation diagram". These diagrams show the points in the sequence $\{x_n, x_{n+1}, x_{n+2}, \ldots\}$ plotted against the parameter b once the orbit has converged to a stable cycle. An example of such a diagram is shown below. This diagram clearly shows that, at certain critical values of b, the behavior of the sequence changes as the period of the orbit doubles.



As a further step towards displaying the interior of the Mandelbrot set, we developed a method for producing similar Feigenbaum diagrams for points in the interior of the set. Given a point $z \in \mathbb{C}$, a set of equally-spaced points on the straight line between the origin and z is generated. For each of these points c, the orbit P_c^{1000} is generated, after which it is assumed that convergence to a periodic cycle has occurred. The distances $|P_c^n - c|$ for the next 1000 points on the orbit are then plotted against $|c|$.

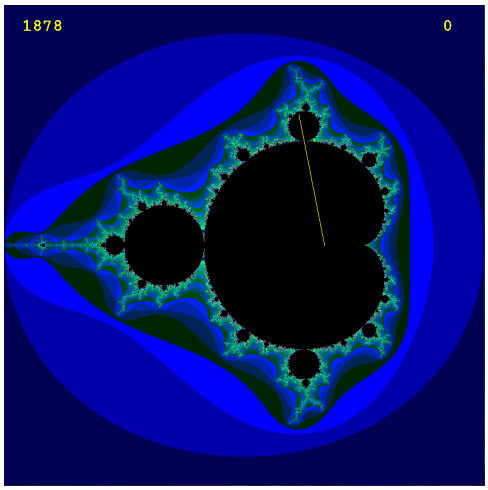
Algorithm 3 Feigenbaum Diagram Algorithm

Require: $w \in \mathbb{C}$
  points = set of equally spaced points from origin to w
  for c in points **do**
    $z = 0$
    for $i = 1$ **to** 1000 **do**
      $z = z^2 + c$
    end for
    for $i = 1$ **to** 1000 **do**
      $z = z^2 + c$
      plot $\|z - c\|$ against $\|c\|$
    end for
  end for

Some representative examples of the diagrams produced by this method are shown below. The image on the left shows the Mandelbrot set and a radial line from the origin to a point in the complex plane. The diagram on the right shows the Feigenbaum diagram produced for the points on this line.

3.2.1 Feigenbaum Diagram for Period 3 Bulb

The Feigenbaum diagram below was produced for a line between the origin and a point in the largest upper period 3 bulb. As the line passes from the main cardioid into the period 3 bulb, it can be seen from the diagram on the right that a "trifurcation" occurs as the orbit changes from one that converges to a single point into one that converges to a cycle of period 3.



Radial line into bulb of period 3 orbits

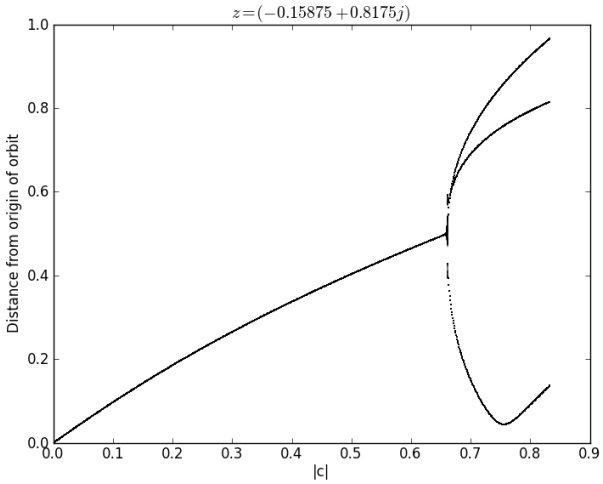
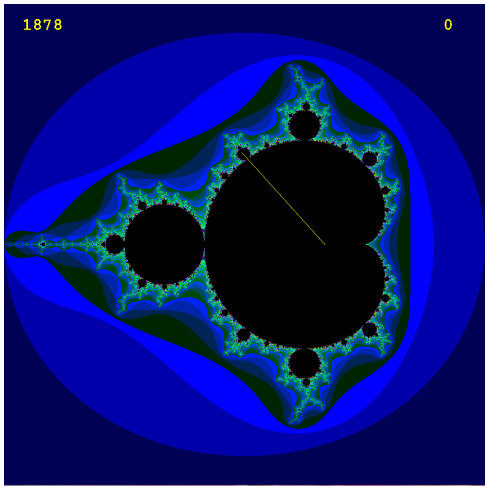


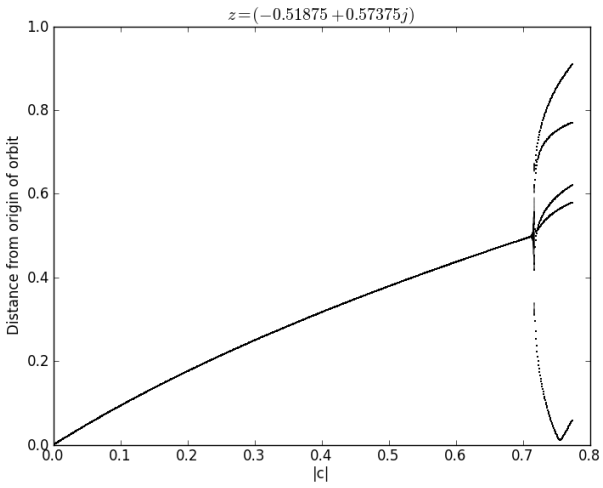
Diagram showing trifurcation inside period 3 bulb

3.2.2 Feigenbaum Diagram for Period 5 Bulb

The Feigenbaum diagram below was produced for a line between the origin and a point in the largest period 5 bulb tangent to the main cardioid on the upper left. As the line passes from the main cardioid into the period 5 bulb, it can be seen that the corresponding line on the Feigenbaum diagram splits into 5. This occurs when the orbit changes from one that converges to a single point into one that converges to a cycle of period 5.



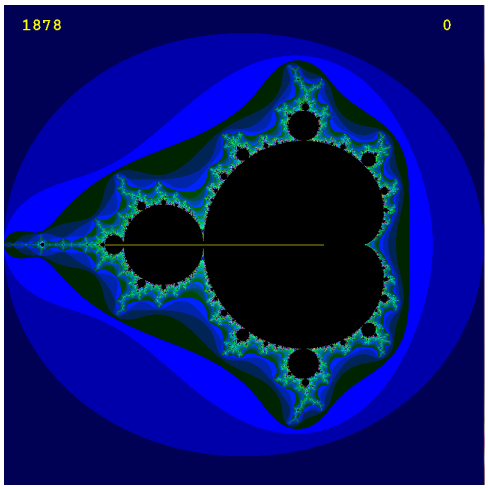
Radial line into bulb of period 5 orbits



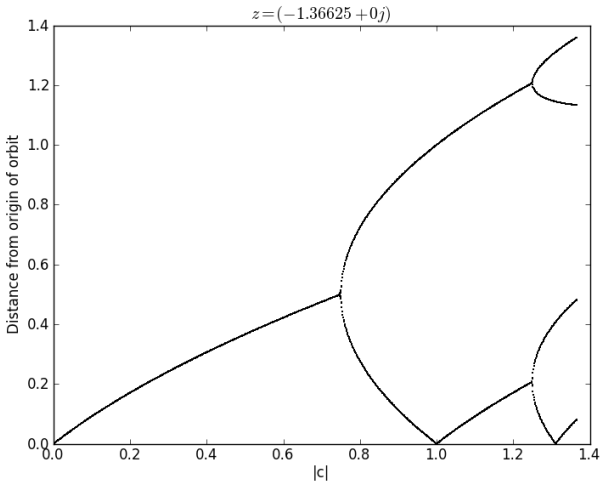
Feigenbaum diagram showing split into 5

3.2.3 Feigenbaum Diagram for Period 2 Bulb, Doubled

The Feigenbaum diagram below was produced for a line between the origin and a point in the secondary period 2 bulb tangent to the large period 2 bulb to the left of the main cardioid. As the line passes from the main cardioid into the period 2 bulb, it can be seen that a "bifurcation" occurs as the orbit changes from one that converges to a single point into one that converges to a cycle of period 2. As the line then passes into the secondary period 2 bulb, a further bifurcation occurs as the orbit changes to one that converges to a cycle of period 4.



Radial line into period 2 bulb



Feigenbaum diagram

3.3 The "Capture-Time" Algorithm: Iterations needed to Converge

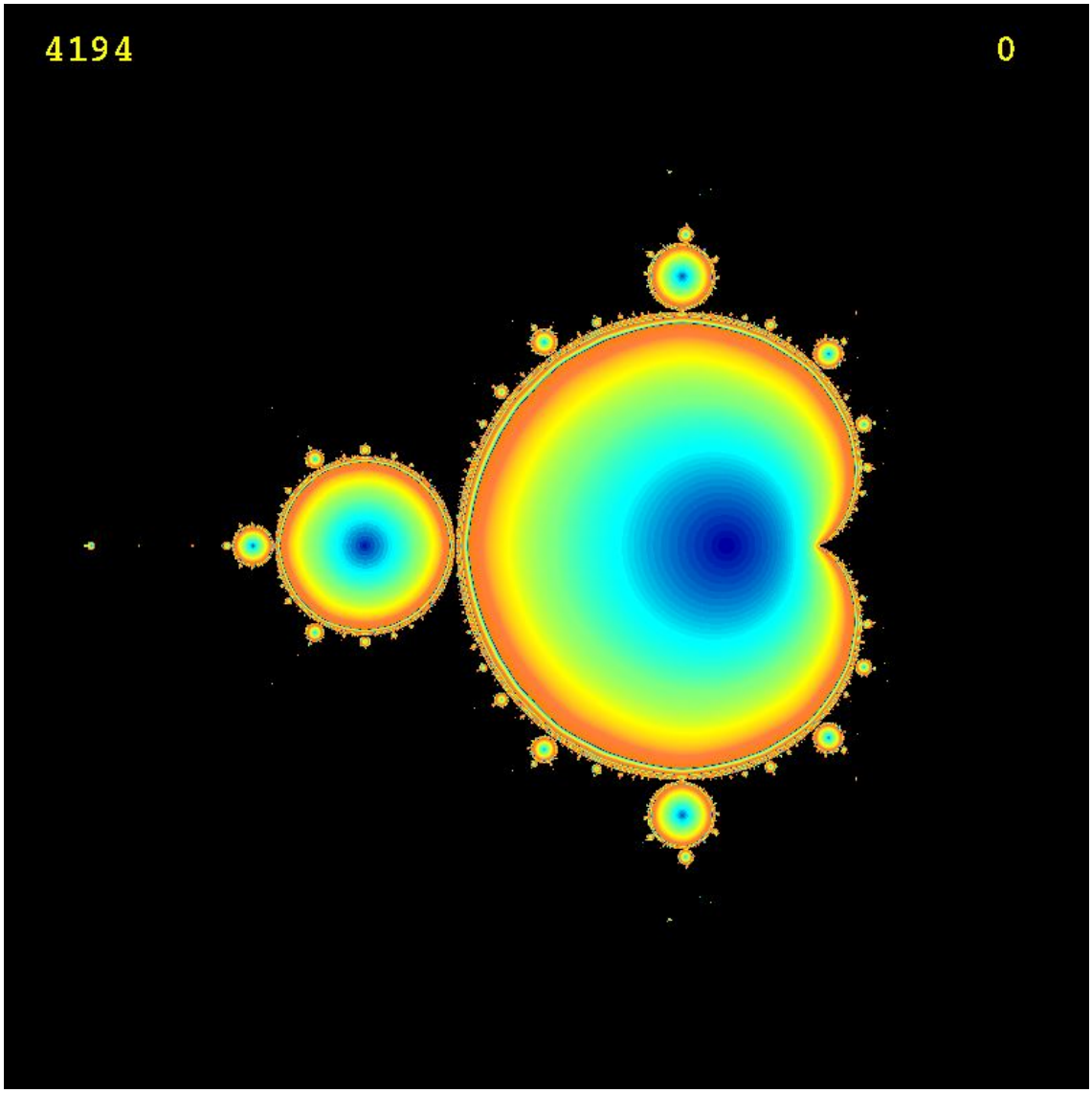
The "capture-time algorithm" is a natural counterpart for points inside the set to the "escape-time algorithm". Given some desired tolerance ϵ, the orbit P_c^n is generated for each point $c \in \mathbb{C}$ until some point in the orbit is closer than ϵ to some previous point in the orbit. The number of iterations needed for this to occur is mapped to a color and displayed at the pixel corresponding to c.

Algorithm 4 Capture-Time Algorithm

Require: $C \in \mathbb{C}$
C = grid of points in a region of the complex plane
Z = C
Image = grid of zero integers
Orbits = [Z]
iteration = 1
loop
$Z = Z^2 + C$
Distance = $|Z - C|$
iteration = iteration + 1
Image[Distance < Closest] = iteration
Closest = minimum(Distance, Closest)
display Image
end loop

3.3.1 Capture-time Map

An image generated using this algorithm is given below.



The following observations can be made:

1. The number of iterations needed to converge is least at the center of the main cardioid and increases towards the edge.
2. The same pattern is seen in the period bulbs tangent to the main cardioid, where the orbits at the center of each bulb converge to a periodic cycle the fastest.

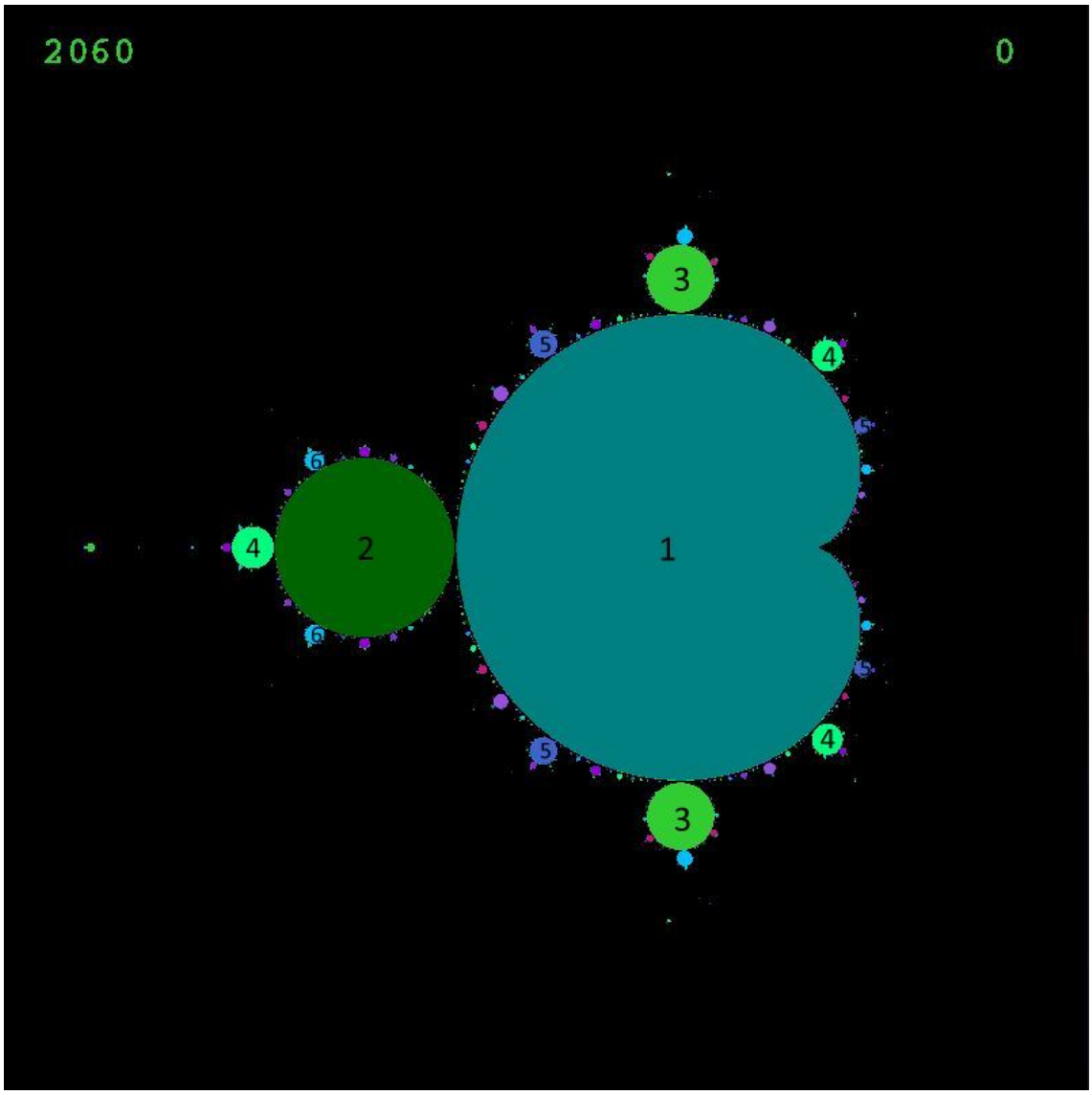
The capture-time algorithm does not however show any features relating to the geometry of the orbits, so better methods to do so are developed in the following sections.

3.4 The "Orbit Cycle" Algorithm: Displaying the Period of the Cycles

The capture-time algorithm can be modified to display the period of the cycle to which orbits converge, instead of the number of iterations needed to converge.

3.4.1 Orbit Cycle Map

An image generated using the "orbit cycle" algorithm is given below, labelled with the period of the cycles to which orbits converge.

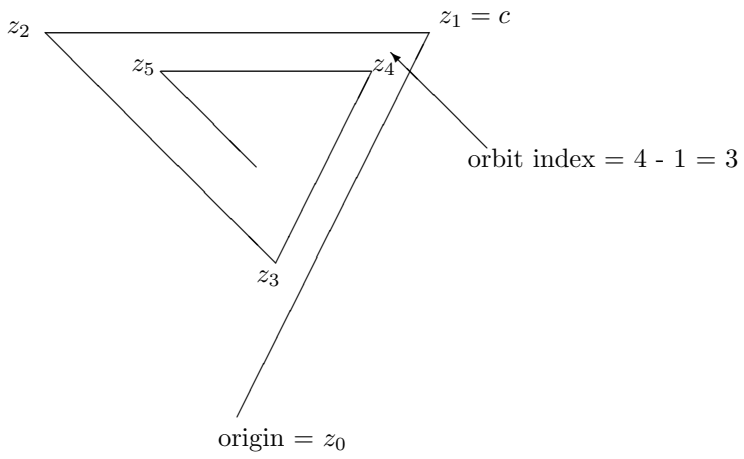


It can be seen that the orbit cycle algorithm partitions the set into distinct components, as described in section 2.3 on the geometry of the Mandelbrot set.

3.5 The "Orbit Index" Algorithm: Displaying the Period of the Orbit

It was seen from the orbit magnitude plots presented in section 3.1 that orbits have a characteristic period associated with them that is different from the period of the cycle to which they converge. As shown previously, the index of the closest point in the orbit P_c^n to the point c provides a way to measure this period.

The orbit index is defined here as one less than the index of the point of minimum distance in the orbit P_c^n from c. The diagram below shows how this index can be interpreted graphically. It can be seen that in this case the orbit most closely resembles a triangle, and that the orbit index is calculated to be three. Similarly, if the shape of the orbit approximates a polygon of a given number of external points, either a simple polygon or a star polygon, then the orbit index will represent the geometry of the shape by a single number. In a following section we will also present a more sophisticated way to capture greater detail about the geometry of an orbit - the "total internal angle" algorithm.

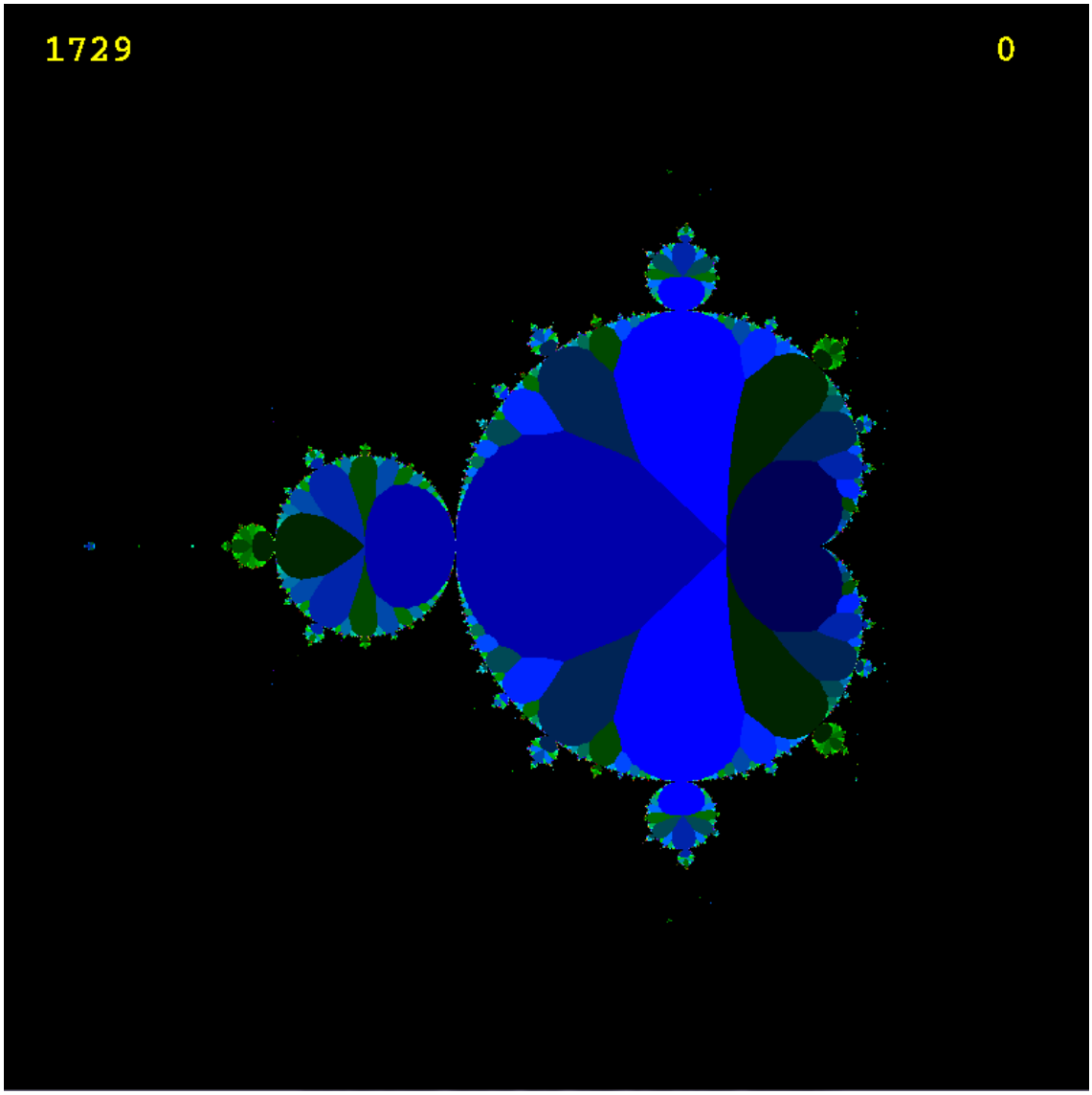


Algorithm 5 Orbit Index Algorithm

Require: $C \in \mathbb{C}$
C = grid of points in a region of the complex plane
Z = C
Image = grid of zero integers
Closest = $|C|$
iteration = 1
loop
$Z = Z^2 + C$
Distance = $|Z - C|$
iteration = iteration + 1
Image[Distance < Closest] = iteration
Closest = minimum(Distance, Closest)
display Image
end loop

3.5.1 Partitioning into Regions of Similar Orbits

An image generated using the orbit index algorithm is given below.

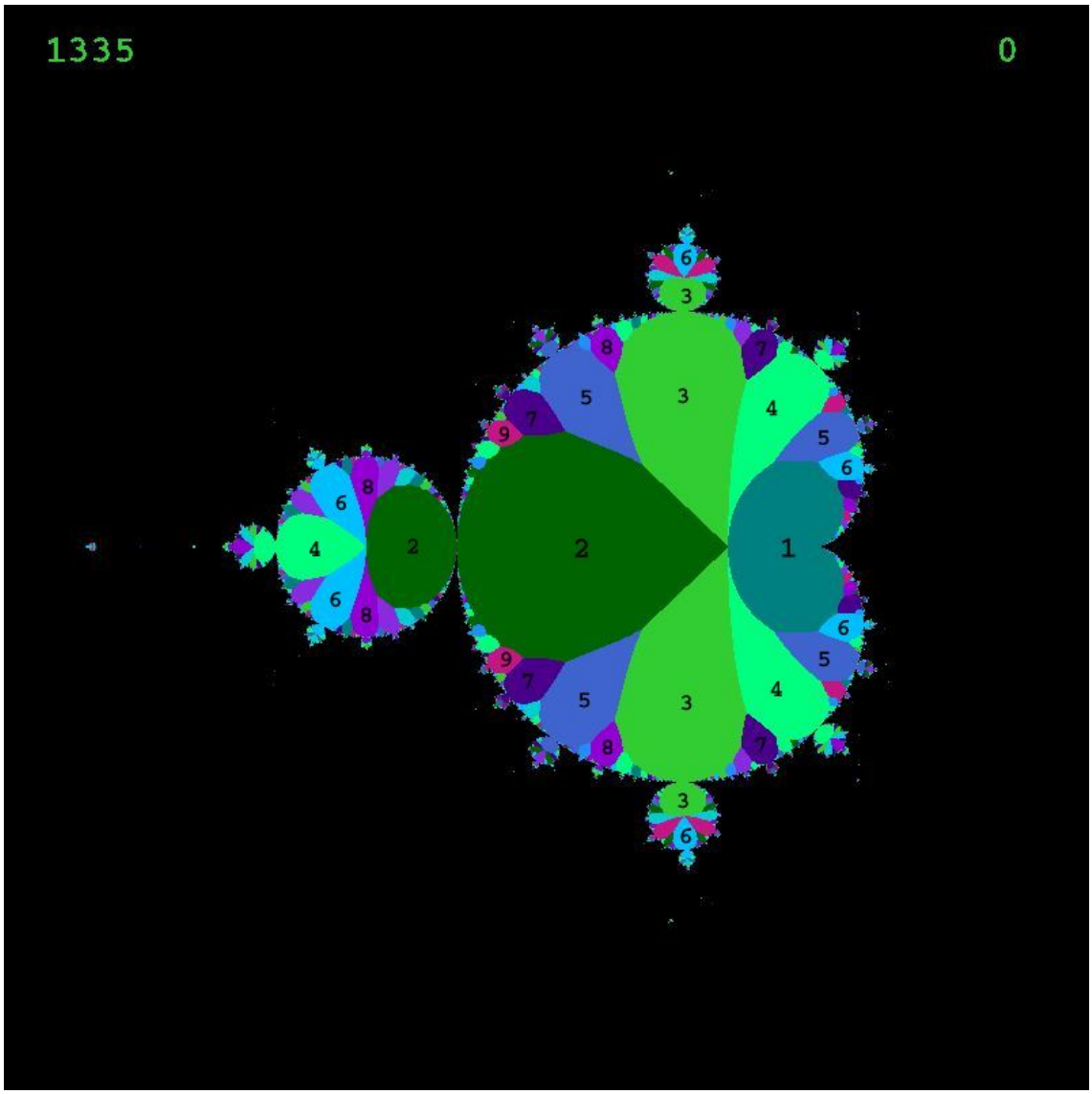


It can be seen immediately that there is an interesting and detailed structure. We can make the following observations:

1. The interior of the Mandelbrot set has been partitioned into distinct regions.
2. The regions span the main cardioid and the period bulbs, including both orbits that converge to a periodic cycle in a bulb with decaying orbits of the same period in the main cardioid.
3. The regions become smaller and more numerous closer to the edge of the main cardioid.
4. The period bulbs show an internal structure similar to that of the main cardioid.

3.5.2 Labelling the Regions

To show this internal structure in greater detail, the following image was produced using a different color map and with the regions labelled with their corresponding indexes.



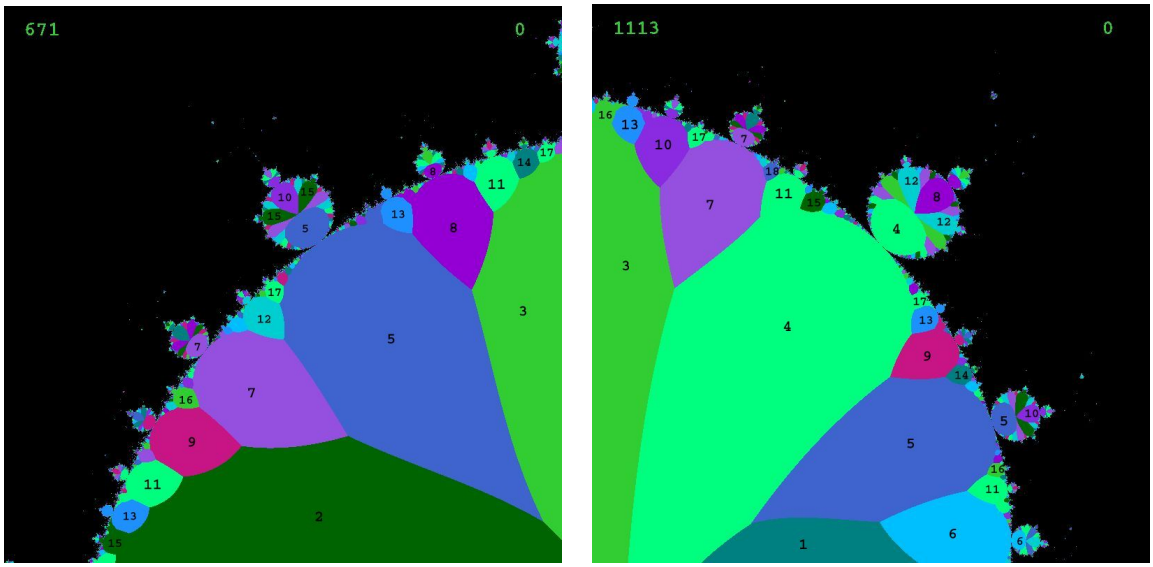
The following observations can now be made:

1. The period of the orbits is greater closer to the edge of the main cardioid.
2. A relationship exists between any two regions and the new region that arises between them closer to the edge of the set. This is referred to as a "Fibonacci" relationship, so called because the period of the new region is the sum of those for the two regions it borders.

3. The period bulbs tangent to the main cardioid are partitioned into similar regions as the main cardioid, but the period of each region is multiplied by the period associated with the bulb. For example, where the main cardioid has regions of period $1, 2, 3, \ldots$, the main period 2 bulb has regions of period $2, 4, 6, \ldots$, the period 3 bulbs have regions of period $3, 6, 9, \ldots$ and so on.

3.5.3 Fibonacci Relationships between Regions

The following two images show the Fibonacci relationships in more detail for the upper left and upper right regions of the main cardioid.



Due to the Fibonacci relationships, it can be seen that bordering each region there exists a sequence of smaller regions whose orbit periods differ by the period of the main region they border. For example, bordering the region of period 2, there exists (going counter-clockwise) a sequence of regions of period $5, 7, 9, 11, 13, \ldots$. Similarly, bordering the region of period 3, there exists (going clockwise) a sequence of regions of period $5, 8, 11, 14, \ldots$.

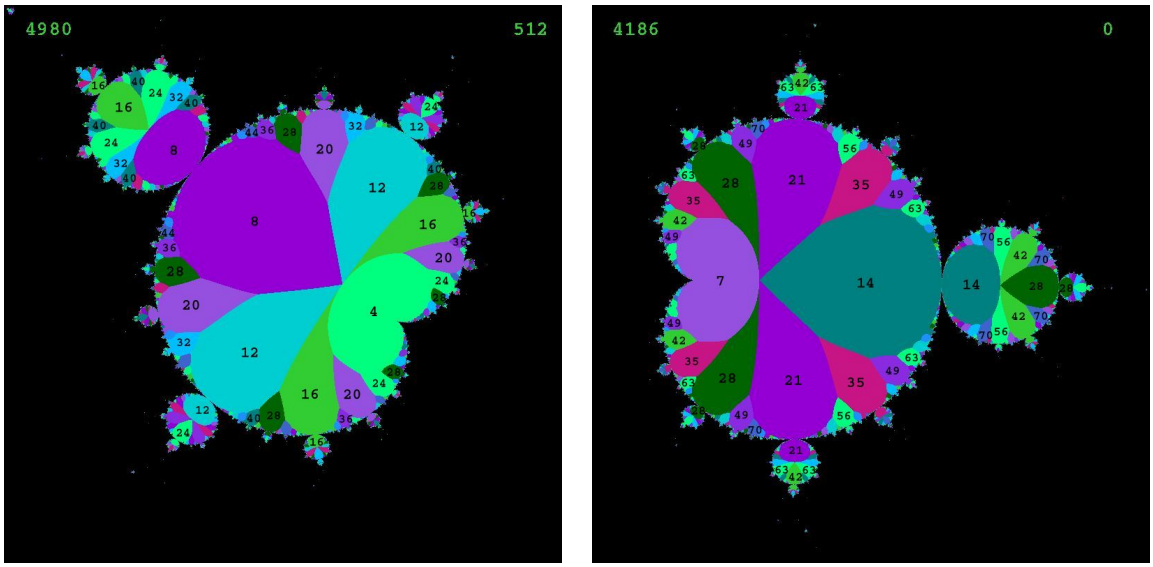
It can also be seen that each region in the main cardioid is associated with one period bulb of the same period, and spans the boundary between the two. We therefore have the following key observation:

The structure of the regions in the orbit index map mirrors the structure of the period bulbs.

Hence, where a Fibonacci relationship exists between regions, an identical Fibonacciship relation exists between the period of the orbits in their associated period bulbs.

3.5.4 Orbit Index Structures in the Satellites

Magnification of the satellite "mini-Mandelbrot sets" surrounding the main body of the Mandelbrot set shows that the structures seen in the main body of the set are duplicated in these satellites on a smaller scale. It can also be seen that, associated with each satellite, there is a characteristic number multiplying each period relative to the corresponding period in the main body of the set. This is demonstrated in the following images.



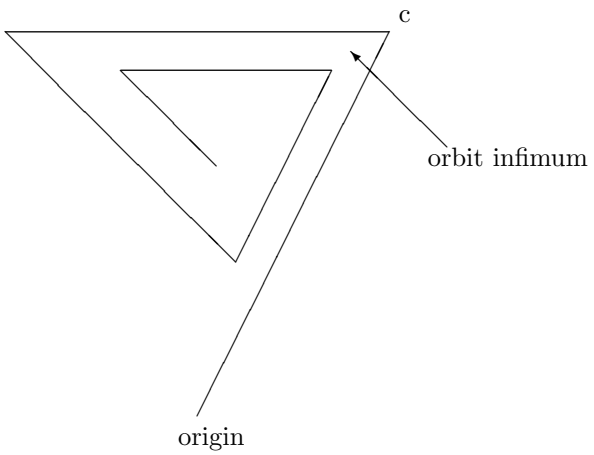
The image on the left above shows the multiplier 4 associated with the satellite. Hence, the index associated with each region is 4 times those seen in the corresponding regions in the main body of the set. Similarly for the image on the right, there is a characteristic number 7 by which the indexes are multiplied.

3.6 The "Orbit Infimum" Algorithm: Estimating Convergence

The orbit index algorithm described in the previous section displayed the index of the closest point in the orbit P_c^n to the point c. Following on from this algorithm, we present a further algorithm which displays instead the minimum distance $|P_c^n - c|$ directly. Since both the period and the decay rate of an orbit are of importance, this is a natural complement to the previous algorithm which showed the period of an orbit. It will be seen that the images produced show how close an orbit is to being a cycle of a fixed integral period.

The diagram below shows the orbit infimum, formally defined as

$$\inf\{|P_c^n - c| \mid n > 1\}$$

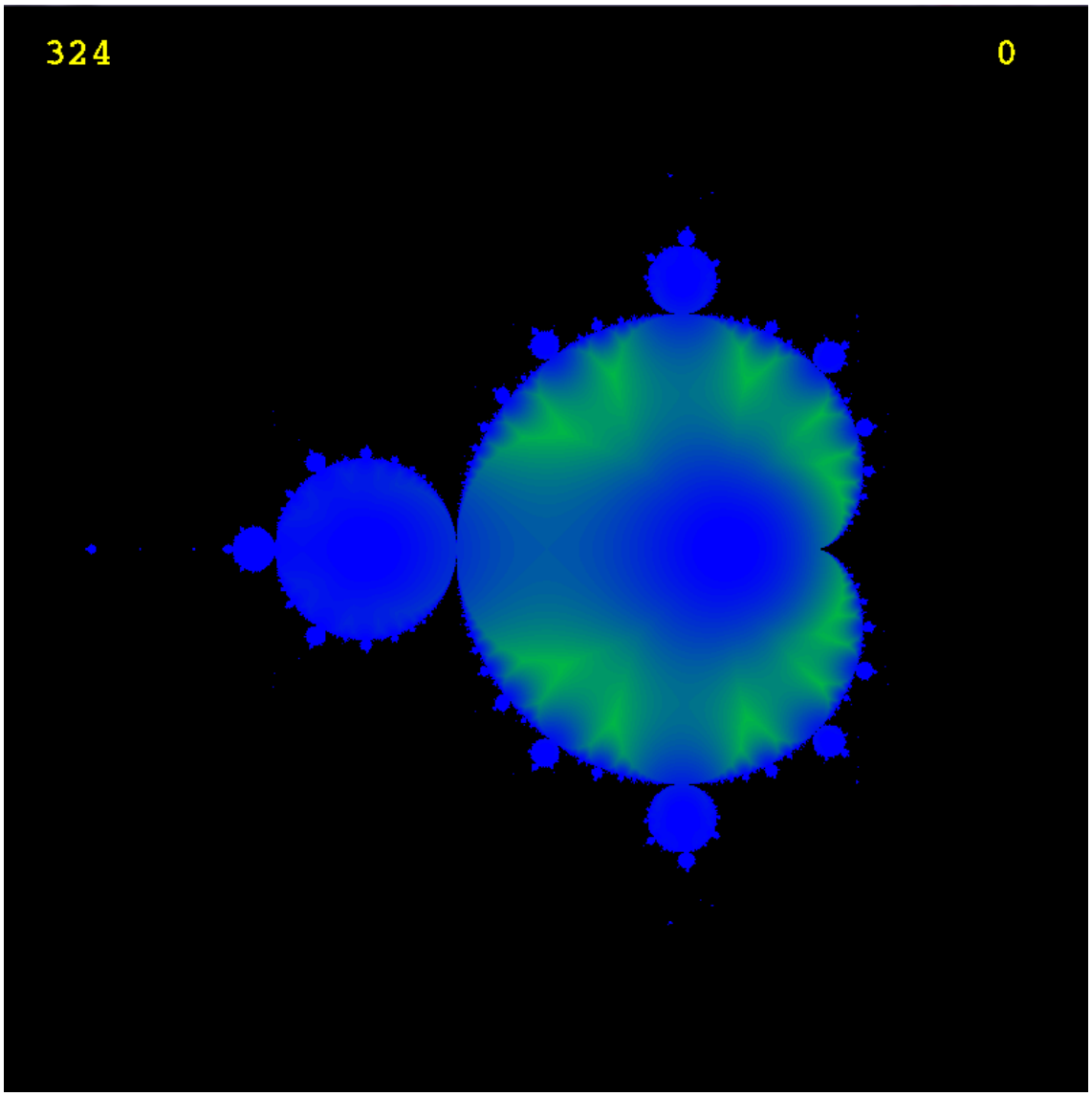


Algorithm 6 Orbit Infimum Algorithm

Require: $C \in \mathbb{C}$
  C = grid of points in a region of the complex plane
  Z = C
  Image = grid of zero integers
  Closest = $|C|$
  loop
    $Z = Z^2 + C$
    Closest = minimum(|Z - C|, Closest)
    Image = Color[Closest]
    display Image
  end loop

3.6.1 Orbit Infimum Map

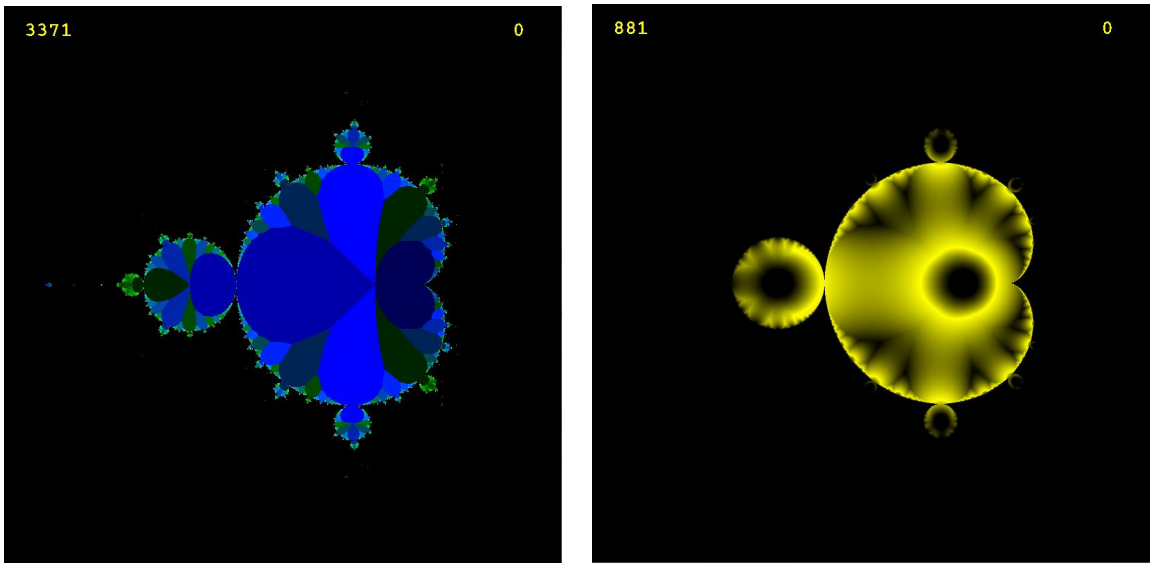
An image generated using the orbit infimum algorithm is given below.



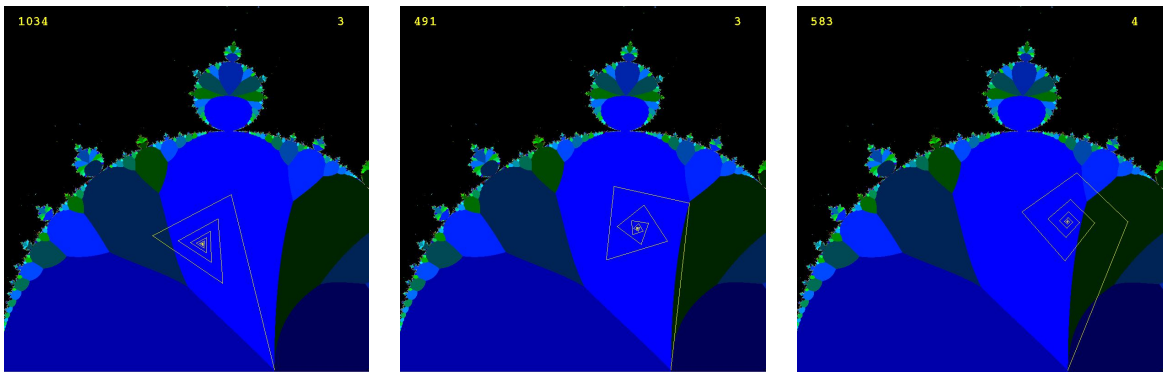
Comparing the image above to those produced by the orbit index algorithm, it can be seen that the lines seen in the image above correspond to boundaries between regions. This is described more fully in the following section.

3.6.2 The Orbit Infimum and the Boundaries between Orbit Index Regions

The following images compare the orbit index on the left with the orbit infimum on the right.



It can be seen that the lines in the orbit infimum image correspond to boundaries in the orbit index image. The reason for this becomes clear if we examine what happens to an orbit when a parameter c crosses the boundary between regions. If $|c|$ is held constant while $\arg c$ is increased, then since multiplication in the complex plane corresponds to the addition of angles, as $\arg c$ increases the internal angles of the orbit will decrease. Visually, it can be seen that the orbit "closes up" when this occurs. This is shown in the following set of images.



Period 3 orbit | Between period 3 and 4 orbit | Period 4 orbit

So, the boundaries in the orbit index map are regions where the orbit infimum is at a local maximum with respect to $\arg c$, as one point in the orbit moves further from c, but the next point in the orbit has not yet come to its closest approach. This manifests as the darker lines between regions in the orbit infimum map. If the orbit infimum were plotted as a surface, this would result in ridges of

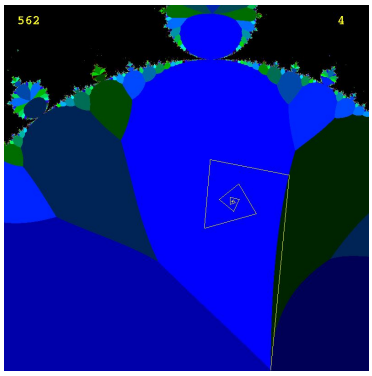
local maxima dividing the regions of the orbit index map.

3.6.3 The Interaction between $|c|$ and $\arg c$ in the Orbit Index Map

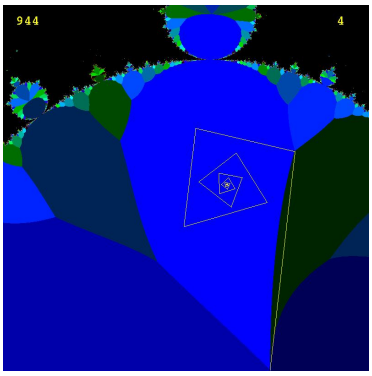
The partitioning of the Mandelbrot set into the regions seen in the orbit index map can also be understood as arising from the interaction between changes in the orbit as $\arg c$ increases (which decreases the internal angles of the orbit) and changes in the orbit as $|c|$ increases (which decreases the rate of convergence to a periodic cycle, as seen in the capture-time map).

This can be seen in the orbit index images below, which show the change in an orbit as $\arg c$ is held approximately constant while $|c|$ is increased along the border between the two largest regions in the main cardioid of period 3 and 4. When c is closer to the origin, $|c|$ is less and the rate of convergence to a single point is greater. As a result, points in the orbit P_c^n for which n is higher converge faster to the fixed point. This is seen in the image below on the left, where the fourth and fifth points P_c^4 and P_c^5 are closer to c than the eights point P_c^8.

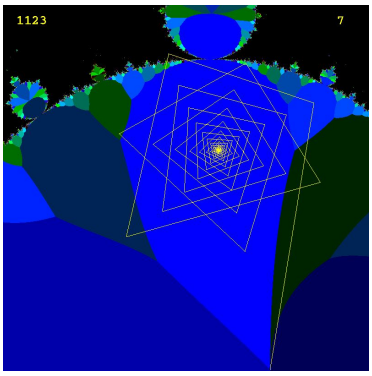
As $|c|$ is increased, the rate of convergence decreases, which alters the relative distances P_c^n. In the center image below, the points P_c^4 and P_c^5 are now equidistant from c with the point P_c^8. In the image below on the right, as $|c|$ is increased further, the point P_c^8 has moved closer to c than either P_c^4 or P_c^5, and the orbit has changed to one of period 7.



Between period 3 and 4 orbit



Between period 3, 4 and 7 orbit



Period 7 orbit

3.7 The "Total Internal Angle" Algorithm

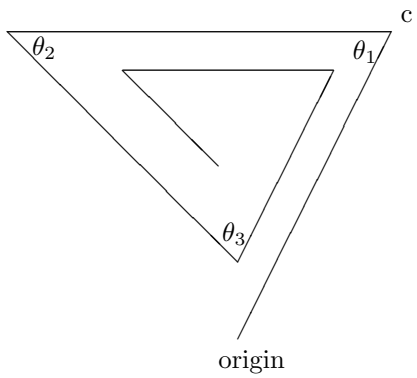
Although the orbit index algorithm does show the period of the orbit, the partitioning of the complex plane into distinct regions corresponding to orbits of different periods has lost some of the information characterizing the orbit. In particular, the closeness of an orbit to a periodic attractor of the same period has been lost.

To recapture some of this information, we developed the "total internal angle" algorithm. The motivation for this algorithm comes from the discussion in the previous section of the way in which the geometry of the orbit changes with $\arg c$ while $|c|$ is held constant. It was shown that, as $\arg c$ increases, the orbit "closes up" as the internal angles of the orbit decrease. Eventually this results in the orbit index changing as a different point in the orbit becomes closer to c than the previous closest point in the orbit.

We would like a way to display these changes in the shape of an orbit *within* a region. The total internal angle algorithm was developed as a way to do this. By summing the internal angles in the first period of an orbit, we have a way to capture the changing shape of an orbit as the internal

angles change. Mapping this angle to a color then provides a way to display the changing shapes of orbits within a given region.

The diagram below shows the total internal angle for an orbit, defined as the sum of the internal angles up to the point of minimum distance in the orbit from c. It can be seen in this example that the orbit most closely resembles a triangle, and that the sum of the internal angles will accordingly be approximately 180°.



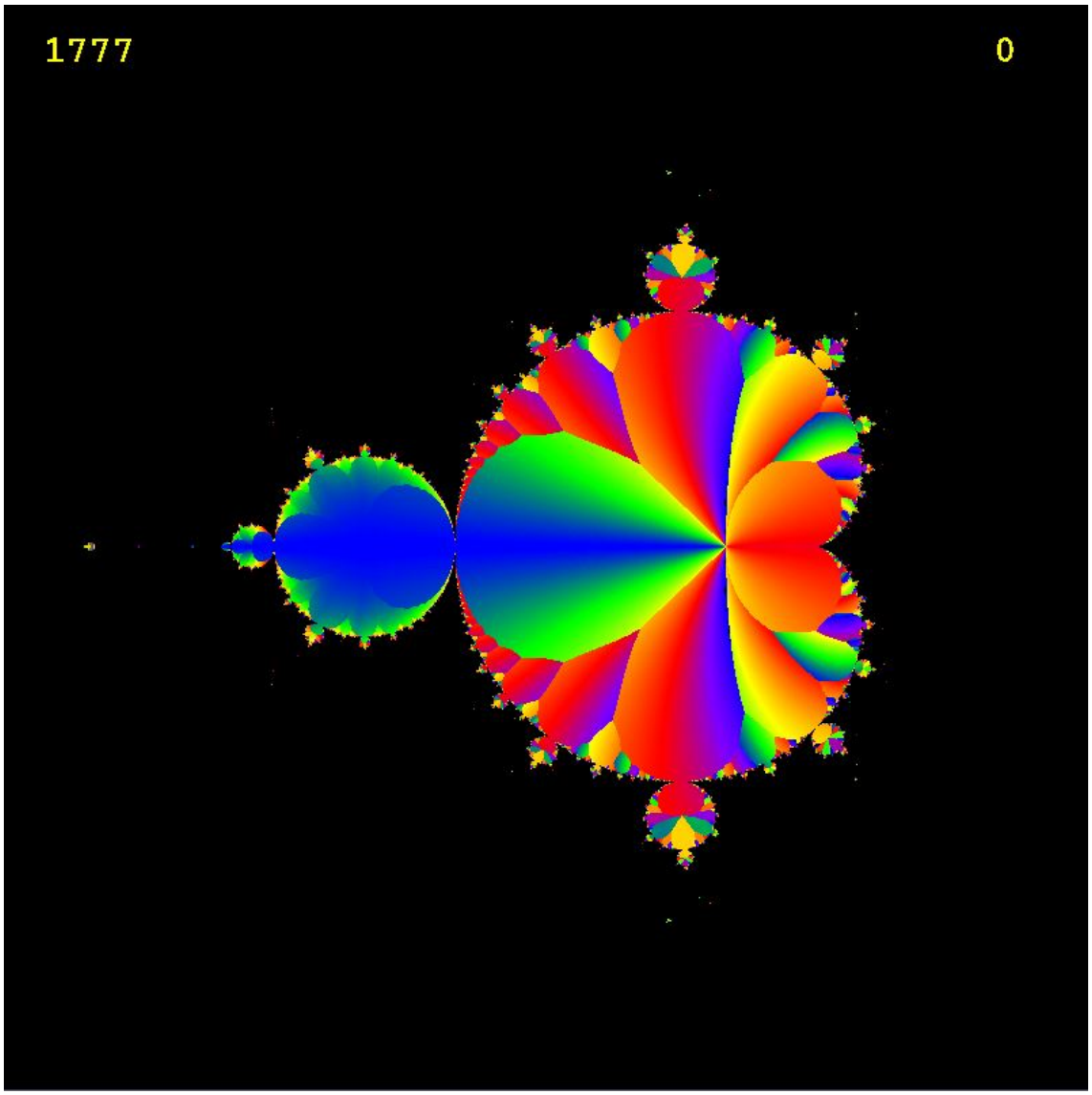
total internal angle $= \theta_1 + \theta_2 + \theta_3 \approx 180°$

Algorithm 7 Total Internal Angle Algorithm

Require: $C \in \mathbb{C}$
  C = grid of points in a region of the complex plane
  $Z = C$
  Previous = grid of complex zeros
  Line = Previous - Z
  Previous = Z
  Angles = grid of float zeros
  Image = grid of integer zeros
  Closest = $|C|$
  loop
    $Z = Z^2 + C$
    Angles = Angles + | arg Line/(Z - Previous)|
    Closer = $|Z - C| <$ Closest
    Image[Closer] = Angles[Closer]
    Closest = minimum($|Z - C|$, Closest)
    display Image
  end loop

3.7.1 The Total Internal Angle Map

The following image was generated using this algorithm.

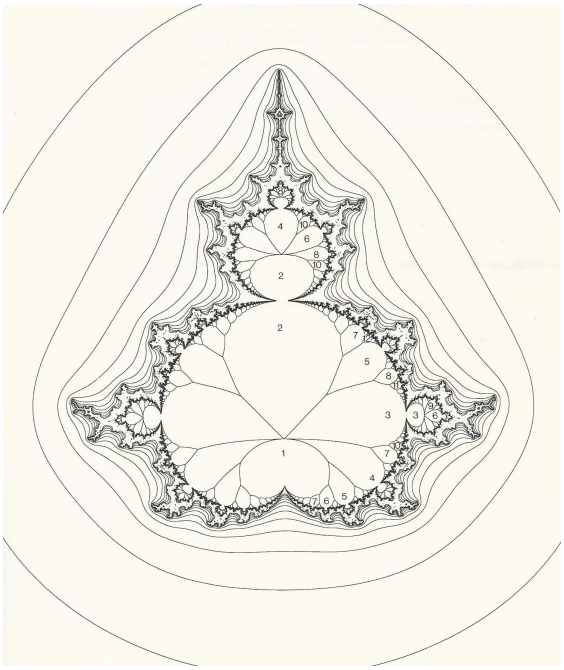


It can be seen that this algorithm does show more detail of the internal structure of regions of the orbit index map, while still preserving the boundaries of the map.

4 Comparison of Results

The images shown in this report are not the first ones that have been presented of the internal structure of the Mandelbrot set. For example, the book "The Beauty of Fractals" [2] contains the following figures:

Beauty of Fractals, Figure 33, page 60



Beauty of Fractals, Figure 34, page 61

These image on the left was produced using

$$\alpha(c) = \inf\{|P_c^k(0)| \mid k = 1, 2, \ldots\}$$

while the image on the right was produced using

$$\text{index}(c) = k \,\text{provided}\, \alpha(c) = |P_c^k(0)$$

The image on the left therefore displays the minimum distance from the origin for any point in the orbit $P_c^k(0)$, while the images on the right shows the value of the index k for which this is so.

It can be seen that the image on the left is similar to that produced in this report using the orbit infimum algorithm, while the image on the right is similar to that produced using the orbit index algorithm. The following analysis shows why this is the case. If we consider successive points in an orbit P_c^n, then we have:

$$\begin{aligned} z_{n+1} &= z_n^2 + c \\ \Rightarrow z_{n+1} - c &= z_n^2 \\ \Rightarrow |z_{n+1} - c| &= |z_n|^2 \\ \Rightarrow \inf\{|z_{n+1} - c|\} &= \inf\{|z_n|^2\} \end{aligned}$$

So, the relationship is this: the index of the point in the orbit which is closest to c is one greater than the index of the point in the orbit that is closest to the origin. This explains the similarity between the two orbit infimum and orbit index images developed for this report and those shown in "The Beauty of Fractals". Although the orbit infimum images were developed with the goal of displaying the underlying period of orbits corresponding to different parameters c, these particular algorithms are therefore similar to those that have been produced before.

5 Errors in Computing the Internal Maps

Before we finish this report, we consider the roundoff errors involved in using floating-point arithmetic.

5.1 Roundoff Error in the Parameter c

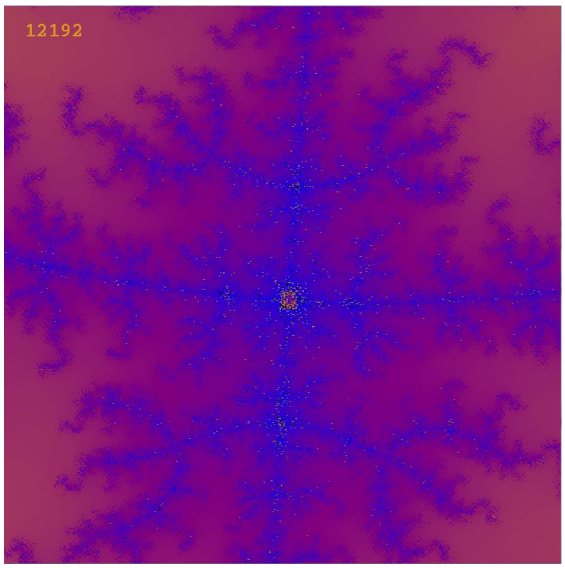
The effects of roundoff error in the parameter c become apparent when the set is repeatedly magnified. As the width of the region being displayed is reduced, the distance between points in the complex plane is corresponding reduced. Beyond a certain magnification depth, neighboring pixels on the screen correspond to points in the complex plane which are separated by a relative distance less than machine epsilon or ϵ_{mach}, where $1 + \epsilon_{mach}$ is the smallest number greater than 1 that can be represented in floating-point format. In the IEEE 754 standard for double precision arithmetic, machine epsilon is 2^{-52}, or approximately 2.22×10^{-16}. For both the real and imaginary parts x and y of the complex parameter c, we therefore have from [3] (where $fl(x)$ denotes the floating point representation of a real number x):

$$\frac{|fl(x) - x|}{|x|} \leq \frac{1}{2}\epsilon_{mach} \approx 1.11 \times 10^{-16} \quad \text{and} \quad \frac{|fl(y) - y|}{|y|} \leq \frac{1}{2}\epsilon_{mach} \approx 1.11 \times 10^{-16}$$

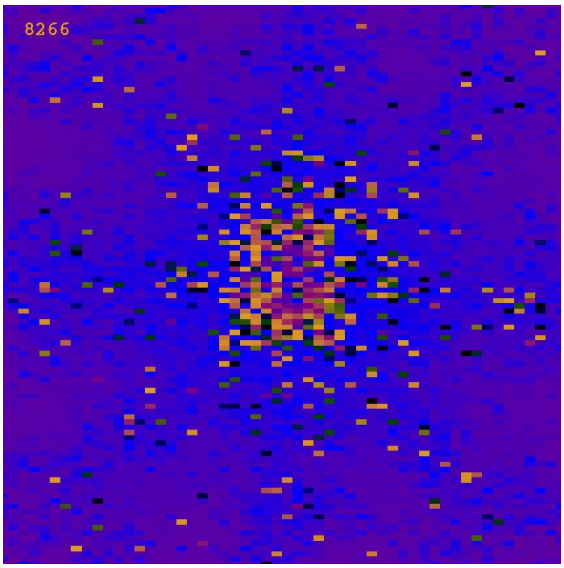
For sufficiently small regions, the roundoff that occurs when c is represented by a pair of finite-precision floating-point numbers results in neighboring pixels on the screen being represented by the same complex floating-point number. The escape-time algorithm will then take the same number of iterations to diverge for both points, so they will be displayed in the same color. The visual effect is that the image loses resolution and becomes segmented into blocks of the same color.

5.1.1 Roundoff Error in the Divergence Map

The effect of roundoff error on the divergence map created using the escape-time algorithm can be seen graphically in the following two images. The image on the left shows a region of the complex plane that is $4.3 * 10^{-14}$ wide. With an 800×800 screen, the pixel separation in this case is approximately $5.4 * 10^{-17}$, which is less than ϵ_{mach}. The image on the right shows a region magnified from the center of the image on the left that is $5.9 * 10^{-15}$ wide, with a pixel separation of approximately $7.4 * 10^{-18}$. The loss of resolution and the division of the image into solid blocks of color can be clearly seen.

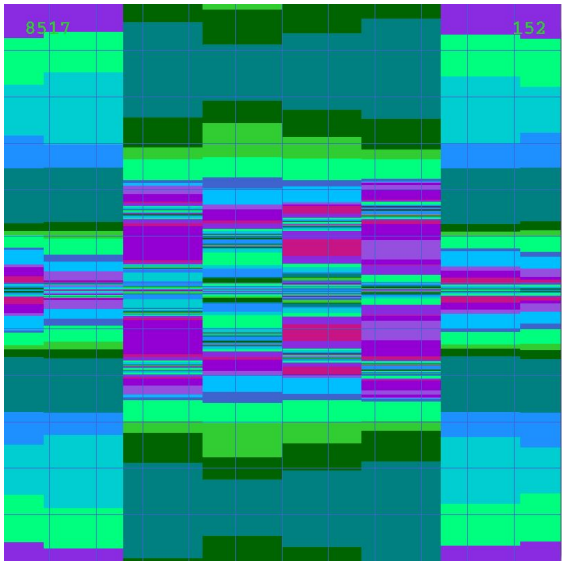


Width $= 4.3 * 10^{-14}$

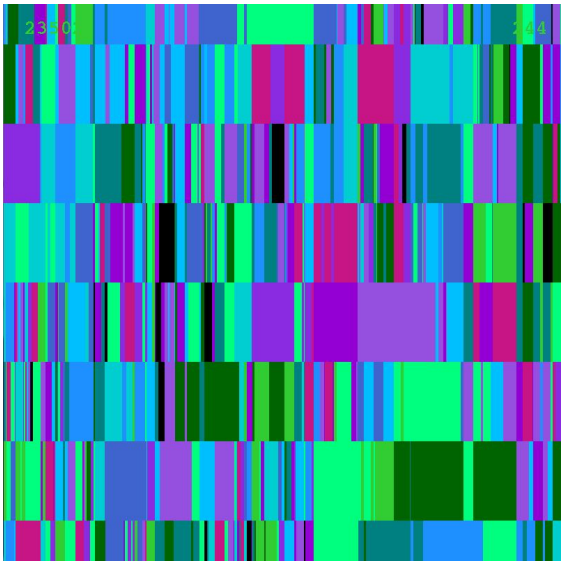


Width $= 5.9 * 10^{-15}$

The fact that roundoff error is relative rather than absolute can be seen in the following images. The image on the left is from a region of the complex plane close to the real axis. The image on the right is from a region close to the imaginary axis. Since the absolute values of the real and imaginary parts of c are significantly different in both of these regions, the effect of roundoff error in these cases is greater along one axis than another. This results in the elongated blocks of color seen in the two images.



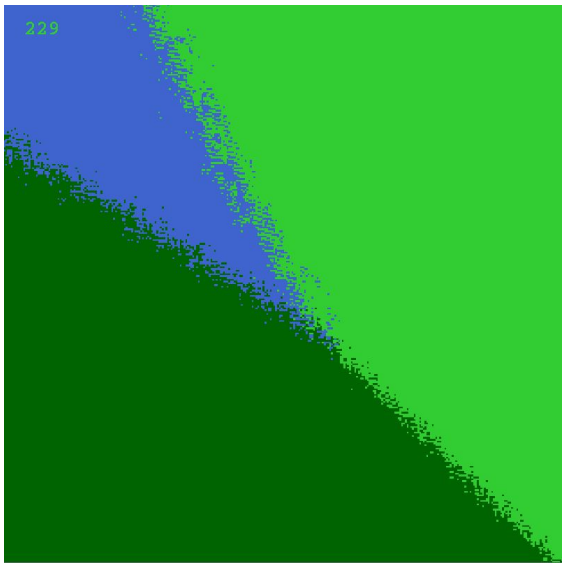
Divergence map at real axis, width $= 1.6 * 10^{-15}$



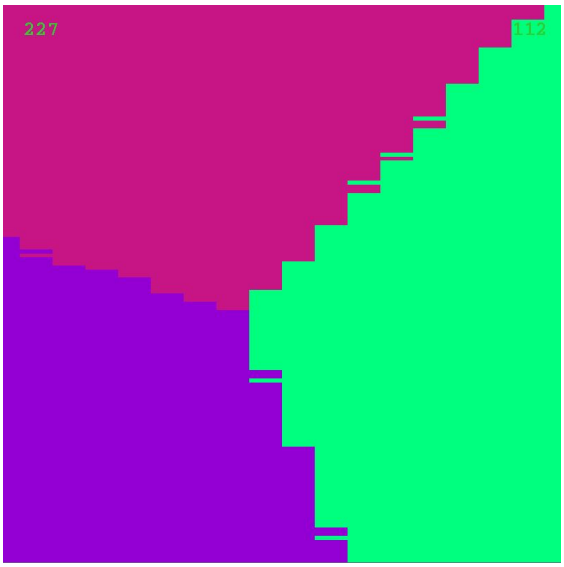
Divergence map at imaginary axis, width $= 1.5 * 10^{-15}$

5.1.2 Roundoff Error in the Orbit Index Maps

Roundoff error in the parameter c will also be an issue when any of the internal maps are magnified. We show below two images from the orbit index map for the main cardioid. The image on the left is from the boundary where regions of period 2, 3 and 5 meet. The image on the right is close to to the edge of the main cardioid where regions of period 61, 112 and 173 meet.



Regions 2, 3, and 5, width $= 8.6 * 10^{-15}$



Regions 61, 112, and 173, width $= 9.6 * 10^{-16}$

It is in some ways surprising how good these images are at this degree of magnification. Despite the relative separation of neighboring pixels being less than machine epsilon, the error seen is roundoff error in the parameter c, rather than accumulated error due to the application of an iterative algorithm.

6 Conclusion

This report described the successful implementation of several different methods for displaying the internal structure of the Mandelbrot set and similarities to existing methods were discussed. In one case, the "total internal angle" method, the images presented are (as far as the author is aware) new images of the internal structure of the set.

Some of the errors involved in displaying the set were also discussed. It was seen that roundoff error in the parameter c prevents magnification of the set beyond a certain point, where the relative separation of pixels representing values in the complex plane approaches machine epsilon.

References

[1] Benoit Mandelbrot *The Fractal Geometry of Nature* 1983: W. H. Freeman and Company

[2] H.-O. Peitgen and P. H. Richter *The Beauty of Fractals* 1986: Springer-Verlag

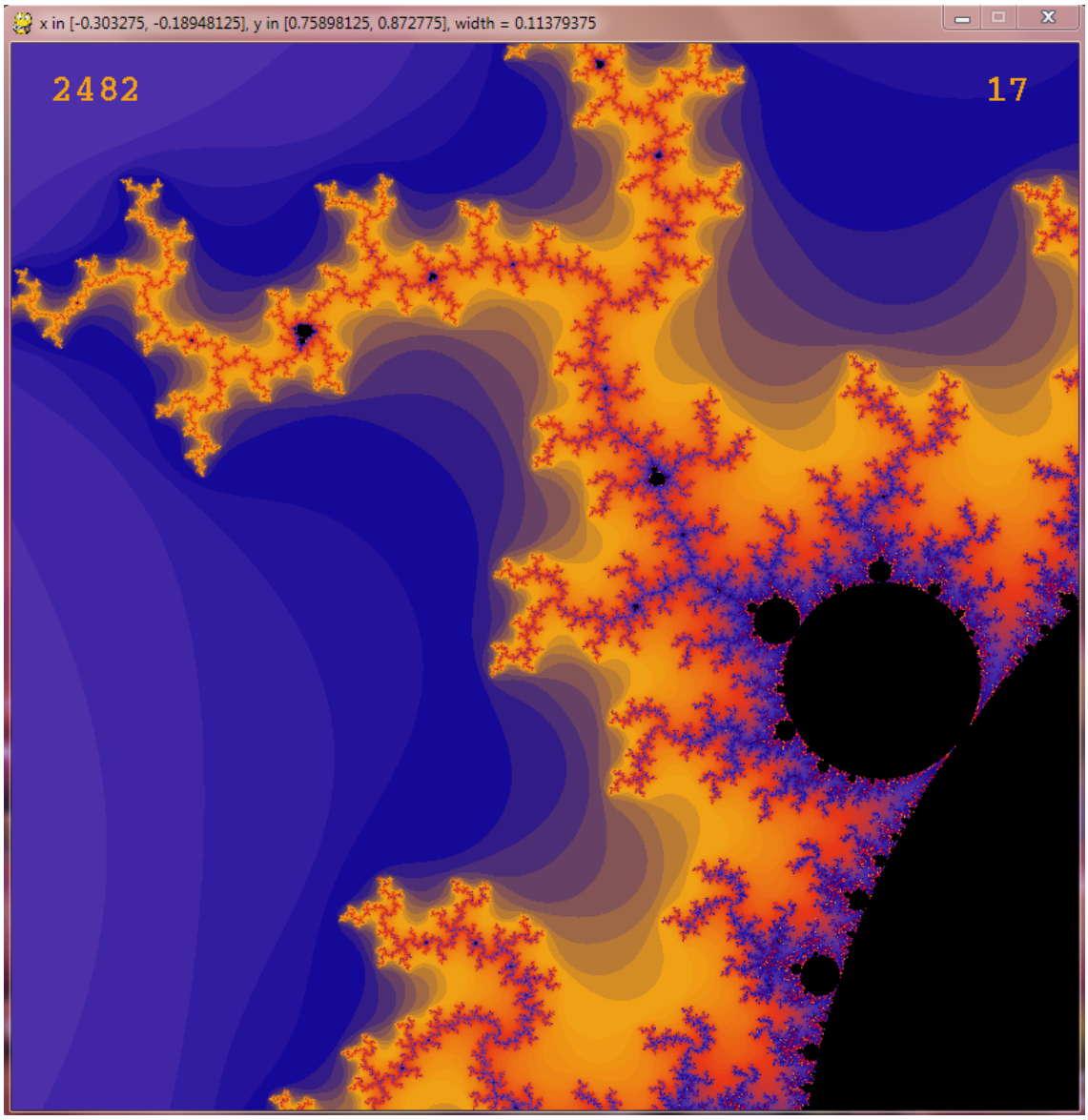
[3] Timothy Sauer *Numerical Analysis, Second Edition* 2012: Pearson

A The "Mandelbrot Explorer" Program

A program was written in Python specifically for this project - the "Mandelbrot Explorer". This program shows a square region of the complex plane in which different maps of the Mandelbrot set can be displayed. The functionality and use of this program is described below.

A.1 The Main Screen

All interaction with "Mandelbrot Explorer" is done through a single main screen, keyboard commands and mouse interaction with the main screen. This screen is shown below.



The user interface has the following elements:

Main Screen This displays one of several different types of image of the Mandelbrot set, described below under "Maps".

Title Bar This shows the region in the complex plane that is being displayed, as well as the width of the region being displayed.

Iteration Number The number in the top left-hand corner displays the current iteration for the map being displayed.

Point number The number in the top right-hand corner displays information associated with the pixel that the cursor is over. Information about any point can therefore be obtained by moving the cursor over the point. The interpretation of this number depends on the type of map being displayed as follows:

Divergence map Number of iterations needed to diverge.

Convergence map Number of iterations needed to converge to a given tolerance.

Cycle map Period of periodic cycle to which orbit converges.

Index map Period of orbit calculated using orbit index.

Infimum map Distance of closest point in orbit to c.

Angle map Sum of internal angles of first periodic orbit (in degrees).

A.2 Program Modes

Mandelbrot Explorer can be run in the following modes.

Magnify Selecting a section of the screen using the mouse results in the selected area being magnified and displayed. The number in the top left corner of the screen shows the number of iterations used for the current image.

Orbit Left mouse clicking on the screen results in the orbit P_c^n being displayed, where c is the complex number corresponding to the point on the screen that was clicked. The number in the top left corner of the screen shows the number of iterations taken for the orbit shown to diverge i.e. the value n such that $|P_c^n| > 2$.

Cycle Color cycles (palette shifts) using the currently selected palette.

A.3 Maps

Six different maps of the Mandelbrot set can be displayed.

Divergence Map Colors pixels by the number of iterations needed to diverge according to the escape-time algorithm.

Convergence Map Colors pixels by the number of iterations needed for points in the Mandelbrot set to converge to an orbit to a given tolerance.

Cycle Map Colors pixels by the period of the periodic cycle to which the orbit converges.

Period Map Colors pixels by the period of the orbit magnitude plot.

Infimum Map Colors pixels by the closest distance to the parameter c attained by any point in an orbit.

Angle Map Colors pixels by the sum of the internal angles of the first period in the orbit magnitude plot.

A.4 Data Analysis

The following types of analysis can be performed.

Distribution Produces a histogram of the number of iterations needed to diverge for the pixels in the current image.

Feigenbaum Diagram Produces a Feigenbaum diagram for points from the origin to where the mouse is clicked.

Orbit Plot Produces an orbit magnitude plot for the point where the mouse is clicked.

A.5 Keyboard Commands

The following keyboard commands are available.

a Switches the map to show the orbit angle.

c Toggles color cycling for the current palette Off - Forward - Backward.

d Switches the map to show the orbit divergence.

f Switch to Feigenbaum mode. Left mouse click then produces a Feigenbaum diagram for the radial line selected.

g Cycles grid type None - Cartesian - Polar

h Switches the map to show orbit convergence.

i Switches the map to show the orbit infimum.

m Switch to magnify mode.

n Toggle image normalization on and off.

o Switch to orbit mode. Left mouse click then produces an orbit plot for the radial line selected.

p Switches the map to show the orbit period.

r Restarts the program at the initial screen in magnify mode.

t Switches the map to show the orbit cycle.

↑ Halve the quantization level on color palettes. Halves the difference needed to show two pixels as a different color.

↓ Double the quantization level on color palettes. Doubles the difference needed to show two pixels as a different color.

SPACE Pauses/unpauses currently active mode.

0 - 9 Select one of ten available palettes, zero being the default palette.

B Conventions used for Algorithm Descriptions

The following conventions are used for the algorithm descriptions in this report:

1. Variable names beginning with lower case letters denote single variables.
2. Variable names beginning with upper case letters denote arrays.
3. Python/Numpy indexing conventions are used for array indexing.

Note that the implementation of these algorithms in the Mandelbrot Explorer involves additional steps for efficiency. At each step, the test for divergence is applied and points that have diverged are removed from further computations. This ensures that we do not perform repeated (and unnecessary) computations for points not in the set. Depending on the particular algorithm, points that have already been classified (by capture-time, orbit index etc) are also removed from further computations.

These steps have been omitted from this report to simplify the descriptions. The code listing included with this report provides full details of the implementation.

C Python Implementation

C.1 Packages Used

Mandelbrot Explorer was written using the following Python packages:

Numpy Supporting fast array processing.

Pygame Providing event handling and animation.

Pylab Data analysis and display.

C.2 Python Code

The Python code written for the Mandelbrot Explorer program created for this report is listed below.

```
# Math 537, Introduction to Numerical Analysis 1, Fall 2013
# Final Project
# File created November 1st, 2013
#-------------------------------------------------------------------------

import pygame
import numpy
from pygame.locals import *
from sys import exit
from random import randint
from numpy import *
from pylab import hist, plot, show, xlabel, ylabel, title
import time

HEIGHT = 800
SCREEN_SIZE = (HEIGHT, HEIGHT)
ORBIT_MODE = 1
MAGNIFY_MODE = 2
CYCLE_MODE = 3
FEIGENBAUM_MODE = 4
DISTANCE_MODE = 5
DIVERGENCE_MAP = 1
HANKEL_MAP = 2
CONVERGENCE_MAP = 3
PERIOD_MAP = 4
ANGLE_MAP = 5
CYCLE_MAP = 6
NO_GRID = 0
CARTESIAN_GRID = 1
POLAR_GRID = 2
EPSILON = 10**(-6)
NUMBER_KEYS = (K_0, K_1, K_2, K_3, K_4, K_5, K_6, K_7, K_8, K_9)

#-------------------------------------------------------------------------
#
# Color maps
#
#-------------------------------------------------------------------------

def make_palette(colors, nodes):
    '''Interpolates colors between given nodes to create a color palette'''
    palette = []
    previous = 0
    for index, n in enumerate(nodes):
        r1, g1, b1 = colors[index]
```

```
        r2, g2, b2 = colors[index + 1]
        rdiff = r2 - r1
        gdiff = g2 - g1
        bdiff = b2 - b1
        gap = n - previous
        for i in range(gap + 1):
            r = int(r1 + i*rdiff/float(gap))
            g = int(g1 + i*gdiff/float(gap))
            b = int(b1 + i*bdiff/float(gap))
            palette += [(r, g, b)]
        previous = n
    return palette

def make_palette_cycle(initial, colors):
    '''Create a color palette by cycling given colors, one per level'''
    ncolors = len(colors)
    palette = [initial]
    for i in range(1, 256):
        color = colors[i%ncolors]
        palette += [color]
    return palette

blackwhite = make_palette([(0,0,0),
                           (255, 255, 255),
                           (255, 255, 255)],
                          [1, 255])
heatmap = make_palette([(0, 0, 0),
                        (0,0,160), # Dark blue
                        (0, 255, 255), # Light blue
                        (128, 255, 128), # Light green
                        (255, 255, 0), # Yellow
                        (255, 128, 64), #Light orange
                        (255, 128, 0)], # Orange
                       [2, 16, 32, 64, 128, 255])
jewel = make_palette([(0, 0, 0),
                      (0, 0, 128),
                      (128, 0, 128),
                      (221, 155, 34),
                      (0, 64, 0),
                      (0, 0, 255),
                      (0, 0, 128),
                      (0, 64, 0),
                      (221, 155, 34),
                      (128, 0, 128),
                      (0, 0, 255)],
                     [2, 4, 6, 8, 12, 16, 32, 64, 128, 255])
redgreen = make_palette([(0, 0, 0),
                         (0, 255, 0),
                         (255, 255, 0),
                         (255, 0, 0)],
```

```
                        [204, 210, 255])
blackyellow = make_palette([(0, 0, 0),
                         (0, 0, 0),
                         (255,255,0),
                         (0, 0, 0)],
                        [10, 64, 255])
redpurple = make_palette([(0, 0, 0),
                          (240, 160, 20),
                          (231, 55, 24),
                          (78,47,170),
                          (22, 9, 151),
                          (240, 160, 20),
                          (231, 55, 24),
                          (78,47,170),
                          (22, 9, 151),
                          (255, 0, 0)],
                         [1, 2, 4, 8, 16, 32, 64, 128, 255])
indi = make_palette([(0, 0, 255),
                     (27, 198, 254),
                     (145, 102, 255),
                     (195, 23, 255),
                     (71, 20, 205)],
                    [64, 128, 192, 255])
indi2 = make_palette([(0, 0, 0),
                      (55, 16, 243),
                      (255, 0, 77),
                      (196, 0, 166),
                      (0, 255, 147),
                      (0, 128, 255)],
                     [2, 32, 64, 128, 255])
rainbow = make_palette([(0, 0, 0),
                        (0, 0, 255), # Blue
                        (0, 255, 0), # Green
                        (255, 255, 0), # Yellow
                        (255, 128, 0), # Orange
                        (255, 0, 0), # Red
                        (128, 0, 255), # Purple
                        (0, 0, 255)], # Blue
                        [1, 43, 85, 127, 170, 213, 255])
ocean = make_palette_cycle((0, 0, 0),
                           [(30, 144, 255),
                            (0, 128, 128),
                            (0, 100, 0),
                            (50, 205, 50),
                            (0, 255, 127),
                            (64, 100, 205),
                            (0, 191, 255),
                            (150, 80, 224),
                            (148, 0, 211),
                            (199, 21, 133),
```

```
                                (138, 43, 226),
                                (0, 255, 127),
                                (0, 206, 209)])

#---------------------------------------------------------------------------
#
# Escape-time algorithm
#
#---------------------------------------------------------------------------

def mandel(n, m, itermax, xmin, xmax, ymin, ymax):
    ix, iy = mgrid[0:n, 0:m]
    x = linspace(xmin, xmax, n)[ix]
    y = linspace(ymin, ymax, m)[iy]
    c = x+complex(0,1)*y
    del x, y
    img = zeros(c.shape, dtype=int)
    ix.shape = n*m
    iy.shape = n*m
    c.shape = n*m
    z = copy(c)
    for i in xrange(itermax):
        # If all points have diverged
        if not len(z): break
        # z(n + 1) = z(n)^2 + c
        multiply(z, z, z)
        add(z, c, z)
        # Check which points have diverged
        rem = abs(z)>2.0
        # Save the iterations needed for the points which just diverged
        img[ix[rem], iy[rem]] = i+1
        # rem is now an array showing the remaining points
        rem = -rem
        # z becomes any points which have not yet diverged
        z = z[rem]
        ix, iy = ix[rem], iy[rem]
        c = c[rem]
    return img

#---------------------------------------------------------------------------
#
# Class to hold all the data
#
#---------------------------------------------------------------------------

class Mandelbrot():

    def __init__(self):
        self.mode = MAGNIFY_MODE
        self.map = DIVERGENCE_MAP
```

```
        self.cycle = True
        self.paused = False

        # Bounding rectangle for the set
        self.xmin = -2.
        self.xmax = 1.
        self.ymin = -1.5
        self.ymax = 1.5
        self.width = 800
        self.height = 800

        # Display coordinates for iteration number and image cursor
        self.text_x = 30
        self.text_y = 20
        self.cursor_x = HEIGHT - 70
        self.cursor_y = 20

        # Magnification rectangle
        self.startx = 0
        self.starty = 0
        self.endx = 0
        self.endy = 0
        self.magnify = False

        # Iterations
        self.itermax = 10000
        self.iter = 0

        # Orbits
        self.orbit = []
        self.zorbit = []
        self.zc = 0

        # Colors
        self.palettes = [blackwhite, heatmap, jewel, rainbow,
                         blackyellow, redpurple, indi, indi2, ocean]
        self.palette_index = 0
        self.palette = heatmap
        self.color = [0, 0, 128]
        self.word_color = [255, 255, 0]
        self.rect_color = [255, 255, 0]
        self.line_color = [255, 255, 0]
        self.grid_color = [127, 127, 127]

        # Display options
        self.levels = 256
        self.normalize = False
        self.grid = NO_GRID
        self.origin = ((HEIGHT * 2)/3, HEIGHT/2)
        self.current_pos = (0, 0)
```

```
        # Fonts
        self.word_font = pygame.font.SysFont("courant", 16, True)
        self.button_font = pygame.font.SysFont("courant", 28, True)

        # image stores the current image
        self.initialize_image()

        pygame.surfarray.use_arraytype('numpy')

    def add_palette(self, palette):
        self.palettes = [palette] + self.palettes
        self.palette = palette

    def initialize_area(self):
        pygame.display.set_caption("Mandelbrot Explorer by Adam Cunningham")
        self.xmin = -2.
        self.xmax = 1.
        self.ymin = -1.5
        self.ymax = 1.5

    #------------------------------------------------------------------------
    # Magnify image functions
    #------------------------------------------------------------------------

    def initialize_image(self):
        self.iter = 0
        n = self.width
        m = self.width
        ix, iy = mgrid[0:n, 0:m]
        x = linspace(self.xmin, self.xmax, self.width)[ix]
        # Array image is stored upside down for correct display
        y = linspace(self.ymax, self.ymin, m)[iy]
        # c holds the complex plane - the parameter space
        c = x+complex(0,1)*y
        del x, y
        # img holds the number of iterations needed for divergence
        img = zeros(c.shape, dtype=int)
        ix.shape = n*m
        iy.shape = n*m
        c.shape = n*m
        self.ix = ix
        self.iy = iy
        self.c = c
        # z holds the results of the latest iteration
        self.z = copy(c)
        # Initialize the internal map - the first closest point is c itself
        self.closest = abs(copy(c))
        self.previous = zeros_like(c)
        self.saved = [self.previous]
```

```
        self.angles = zeros(c.shape, dtype=float)
        self.img = img

    #---------------------------------------------------------------------
    # Functions for updating the image using different maps
    #---------------------------------------------------------------------

    def update(self):
        ''' Perform one iteration of the currently active mode'''
        if self.paused == True:
            return
        elif self.mode == MAGNIFY_MODE:
            if self.map == DIVERGENCE_MAP:
                self.update_divergence()
            elif self.map == HANKEL_MAP:
                self.update_hankel()
            elif self.map == CONVERGENCE_MAP:
                self.update_internal()
            elif self.map == PERIOD_MAP:
                self.update_period()
            elif self.map == ANGLE_MAP:
                self.update_angle()
            elif self.map == CYCLE_MAP:
                self.update_time()
        elif self.mode == ORBIT_MODE:
            self.update_orbit()
        elif self.mode == CYCLE_MODE:
            self.update_cycle()

    def update_divergence(self):
        '''Updates the divergence map by one iteration'''
        z = self.z
        c = self.c
        i = self.iter
        img = self.img
        ix = self.ix
        iy = self.iy
        # Update z to the next point in the orbit
        multiply(z, z, z)
        add(z, c, z)
        # Check which points have diverged
        rem = abs(z)>2.0
        # Save the iterations needed for the points which just diverged
        img[ix[rem], iy[rem]] = i+1
        # rem is now an array showing the remaining points
        rem = -rem
        # z becomes any points which have not yet diverged
        self.z = z[rem]
        self.ix, self.iy = ix[rem], iy[rem]
        self.c = c[rem]
```

```
        self.iter += 1
        self.img = img

    def update_convergence(self):
        '''Updates the convergence map by one iteration'''
        z = self.z
        c = self.c
        i = self.iter
        img = self.img
        ix = self.ix
        iy = self.iy
        closest = self.closest
        multiply(z, z, z)
        add(z, c, z)
        # Find the distance of the latest point in orbit from c
        distance = abs(c - z)
        closer = distance < closest
        # If latest point in orbit is closer than previous, update image
        img[ix[closer], iy[closer]] = i + 1
        # Save the new minimum distance
        minimum(distance, closest, closest)
        # Check which points have diverged
        rem = abs(z)>2.0
        # Points which just diverged get set to zero in the image
        img[ix[rem], iy[rem]] = 0
        # rem is now an array showing the remaining points
        rem = -rem
        # z becomes any points which have not yet diverged
        self.z = z[rem]
        self.ix, self.iy = ix[rem], iy[rem]
        self.c = c[rem]
        self.closest = closest[rem]
        self.iter += 1
        self.img = img

    def update_hankel(self):
        '''Updates the hankel map by one iteration'''
        z = self.z
        previous = copy(z)
        c = self.c
        i = self.iter
        saved = self.saved
        img = self.img
        ix = self.ix
        iy = self.iy
        # Update z to the next point in the orbit
        multiply(z, z, z)
        add(z, c, z)
        # Check which points have diverged
        rem1 = abs(z)>2.0
```

```
        # Check which points have converged
        rem2 = abs(previous - z) < EPSILON
        # Compare with all previous points in orbit
        for s in saved:
            rem3 = abs(s - z) < EPSILON
            rem2 = logical_or(rem2, rem3)
        rem = logical_or(rem1, rem2)
        # Any points which have diverged get set to zero in the image
        img[ix[rem1], iy[rem1]] = 0
        # Any points which have converged get set to i + 1 in the image
        img[ix[rem2], iy[rem2]] = i + 1
        # rem is now an array showing the remaining points
        rem = -rem
        # Update z and c to only include points which have not yet diverged
        self.z = z[rem]
        self.ix, self.iy = ix[rem], iy[rem]
        self.c = c[rem]
        for j, s in enumerate(saved):
            saved[j] = s[rem]
        self.saved = saved + [copy(self.z)]
        self.iter += 1
        self.img = img

    def update_time(self):
        '''Updates the cycle map by one iteration'''
        z = self.z
        previous = copy(z)
        c = self.c
        i = self.iter
        saved = self.saved
        img = self.img
        ix = self.ix
        iy = self.iy
        # Update z to the next point in the orbit
        multiply(z, z, z)
        add(z, c, z)
        # Check which points have diverged
        rem1 = abs(z)>2.0
        # Check which points have converged
        rem2 = abs(previous - z) < EPSILON
        rem = logical_or(rem1, rem2)
        img[ix[rem2], iy[rem2]] = 1
        # Compare with all previous points in orbit
        for j, s in enumerate(saved):
            rem3 = abs(s - z) < EPSILON
            # Points which converge get set to the period of the cycle
            img[ix[rem3], iy[rem3]] = i - j + 1
            rem2 = logical_or(rem2, rem3)
        #rem = logical_or(rem1, rem2)
        #rem = rem1
```

```
        # Any points which have diverged get set to zero in the image
        img[ix[rem1], iy[rem1]] = 0
        # rem is now an array showing the remaining points
        rem = -rem
        # Update z and c to only include points which have not yet diverged
        self.z = z[rem]
        self.ix, self.iy = ix[rem], iy[rem]
        self.c = c[rem]
        for j, s in enumerate(saved):
            saved[j] = s[rem]
        self.saved = saved + [copy(self.z)]
        self.iter += 1
        self.img = img

    def update_internal(self):
        '''Updates the internal map by one iteration'''
        z = self.z
        c = self.c
        i = self.iter
        img = self.img
        ix = self.ix
        iy = self.iy
        closest = self.closest
        # Update z to the next point in the orbit
        multiply(z, z, z)
        add(z, c, z)
        # Save minimum distance of any point from c
        minimum(abs(z - c), closest, closest)
        # Check which points have diverged
        rem = abs(z)>2.0
        # Points which just diverged get set to zero in the image
        img[ix[rem], iy[rem]] = 0
        # rem is now an array showing the remaining points
        rem = -rem
        # z becomes any points which have not yet diverged
        self.z = z[rem]
        self.ix, self.iy = ix[rem], iy[rem]
        self.c = c[rem]
        self.closest = closest[rem]
        img[ix[rem], iy[rem]] = ceil(self.closest * self.levels)
        self.iter += 1
        self.img = img

    def update_period(self):
        '''Updates the period map by one iteration'''
        z = self.z
        c = self.c
        i = self.iter
        img = self.img
        ix = self.ix
```

```
        iy = self.iy
        closest = self.closest
        multiply(z, z, z)
        add(z, c, z)
        # Find the distance of the latest point in orbit from c
        distance = abs(c - z)
        closer = distance < closest
        # If latest point in orbit is closer than previous, update image
        img[ix[closer], iy[closer]] = i + 1
        # Save the new minimum distance
        minimum(distance, closest, closest)
        # Check which points have diverged
        rem = abs(z)>2.0
        # Points which just diverged get set to zero in the image
        img[ix[rem], iy[rem]] = 0
        # rem is now an array showing the remaining points
        rem = -rem
        # z becomes any points which have not yet diverged
        self.z = z[rem]
        self.ix, self.iy = ix[rem], iy[rem]
        self.c = c[rem]
        self.closest = closest[rem]
        self.iter += 1
        self.img = img

    def update_angle(self):
        '''Updates the angle map by one iteration'''
        z = self.z
        c = self.c
        i = self.iter
        img = self.img
        ix = self.ix
        iy = self.iy
        closest = self.closest
        previous = self.previous
        angles = self.angles
        previous_line = previous - z
        previous[:] = z
        # Update z to the next point in the orbit
        multiply(z, z, z)
        add(z, c, z)
        # Keep cumulative total of the angles in the orbit
        angles += abs(angle(previous_line/(z - previous), deg=True))
        # Find the distance of the latest point in orbit from c
        distance = abs(c - z)
        closer = distance < closest
        # If latest point in orbit is closer than previous, update image
        # Show sum of interior angles of orbit
        img[ix[closer], iy[closer]] = maximum(1, rint(angles[closer]))
        # Save the new minimum distance
```

```
        minimum(distance, closest, closest)
        # Check which points have diverged
        rem = abs(z)>2.0
        # Points which just diverged get set to zero in the image
        img[ix[rem], iy[rem]] = 0
        # rem is now an array showing the remaining points
        rem = -rem
        # z becomes any points which have not yet diverged
        self.z = z[rem]
        self.ix, self.iy = ix[rem], iy[rem]
        self.c = c[rem]
        self.closest = closest[rem]
        self.previous = previous[rem]
        self.angles = angles[rem]
        self.iter += 1
        self.img = img

    def update_cycle(self):
        ''' Perform one iteration of currently active mode'''
        palette = self.palette
        # If cycling forward
        if self.cycle:
            if isinstance(palette, tuple):
                self.palette = (palette[-1],) + palette[:-1]
            else:
                self.palette = [palette[-1]] + palette[:-1]
        # Else must be cycling backward
        else:
            if isinstance(palette, tuple):
                self.palette = palette[1:] + (palette[1],)
            else:
                self.palette = palette[1:] + [palette[1]]

    #--------------------------------------------------------------------------
    # Render the image and associated information
    #--------------------------------------------------------------------------

    def render(self, surface):
        # Draw the latest image to the surface
        if self.normalize:
            # Normalize the image
            image = copy(self.img)
            m = image.max()
            image *= 255
            image /= m
            pygame.surfarray.blit_array(surface, image)
        else:
            pygame.surfarray.blit_array(surface, self.img)
        # Draw the magnifying rectangle
        if self.magnify == True:
```

```
            r = Rect(self.startx, self.starty,
                        self.endx - self.startx, self.endy - self.starty)
            pygame.draw.rect(surface, self.rect_color, r, 1)
        # Draw the overlying grid
        if self.grid == CARTESIAN_GRID:
            intervals = 12
            for i in arange(1, intervals):
                x = (i * HEIGHT)/intervals
                pygame.draw.line(surface, self.grid_color, (x, 0), (x, HEIGHT))
                pygame.draw.line(surface, self.grid_color, (0, x), (HEIGHT, x))
        elif self.grid == POLAR_GRID:
            origin_x, origin_y = self.origin
            # Draw the lines of constant radius
            for i in arange(1, 10):
                radius = (i * HEIGHT)/12
                pygame.draw.circle(surface, self.grid_color, self.origin, radius, 1)
            # Draw the lines of constant angle
            lines = 6
            radius = (HEIGHT * 2)/3
            pygame.draw.line(surface, self.grid_color, (0, origin_y), (HEIGHT, origin_y))
            for i in arange(1, lines):
                theta = (i*pi)/lines
                end_y = origin_y + (radius* sin(theta))
                end_x = origin_x + (radius* cos(theta))
                pygame.draw.line(surface, self.grid_color, self.origin,
                                 (end_x, end_y))
                end_y = origin_y - (radius* sin(theta))
                pygame.draw.line(surface, self.grid_color, self.origin,
                                 (end_x, end_y))
        # Update the iteration number on the screen
        text_surface = self.button_font.render(str(self.iter), False, self.word_color)
        surface.blit(text_surface, (self.text_x, self.text_y))
        # Update the cursor image number on the screen
        pixel_value = self.img[self.current_pos[0], self.current_pos[1]]
        text_surface = self.button_font.render(str(pixel_value), False, self.word_color)
        surface.blit(text_surface, (self.cursor_x, self.cursor_y))
        # Draw the orbit
        if self.mode == ORBIT_MODE and (len(self.orbit) > 1):
            pygame.draw.lines(surface, self.line_color, False, self.orbit)
        # Draw the line along which the Feigenbaum disgram is created
        if self.mode == FEIGENBAUM_MODE:
            pygame.draw.line(surface, self.line_color, self.origin, self.current_pos)
        # If the palette is being cycled, make sure it shows up
        if self.mode == CYCLE_MODE:
            screen.set_palette(self.palette)

    #--------------------------------------------------------------------
    # Magnify mode
    #--------------------------------------------------------------------
```

```
    def start_magnify(self, x, y):
        '''Save the start of a magnify rectangle'''
        self.magnify = True
        self.startx = x
        self.starty = y
        self.endx = x
        self.endy = y

    def end_magnify(self, x, y):
        if self.magnify == False:
            return
        self.magnify = False

        # Dimensions of the area to magnify
        left = min(x, self.startx)
        right = max(x, self.startx)
        top = min(y, self.starty)
        bottom = max(y, self.starty)

        pixel_width = min(right - left, bottom - top)
        if pixel_width < 2:
            self.magnify = False
            return
        old_width = self.xmax - self.xmin
        new_width = pixel_width * old_width/800.

        # Update the new area of the image
        self.xmin = self.xmin + (left * old_width/800.)
        self.xmax = self.xmin + new_width
        self.ymax = self.ymax - (top * old_width/800.)
        self.ymin = self.ymax - new_width
        pygame.display.set_caption('x in ['
                                   + str(self.xmin) + ', '
                                   + str(self.xmax) + '], y in ['
                                   + str(self.ymin) + ', '
                                   + str(self.ymax) + '],'
                                   + ' width = ' + str(new_width))
        self.initialize_image()

    def move_magnify(self, x, y):
        '''Update the magnifying rectangle'''
        if self.magnify == False:
            return
        left = min(x, self.startx)
        right = max(x, self.startx)
        top = min(y, self.starty)
        bottom = max(y, self.starty)
        width = min(right - left, bottom - top)
        # Restrict the magnifying rectangle to be a square
        self.startx = left
```

```
        self.starty = top
        self.endx = left + width
        self.endy = top + width
        return

    #---------------------------------------------------------------------------
    # Conversions between screen coordinates and points in complex plane
    #---------------------------------------------------------------------------

    def xy_to_z(self, x, y):
        '''Convert screen coordinates into a complex number'''
        width = self.xmax - self.xmin
        zx = self.xmin + x * width/800.
        zy = self.ymax - y * width/800.
        return complex(zx, zy)

    def z_to_xy(self, z):
        '''Convert complex number into screen coordinates'''
        width = self.xmax - self.xmin
        zx = real(z)
        zy = imag(z)
        x = int(((zx - self.xmin) * 800.)/width)
        y = int(((self.ymax - zy) * 800.)/width)
        return (x, y)

    #---------------------------------------------------------------------------
    # Orbit mode
    #---------------------------------------------------------------------------

    def initialize_orbit(self):
        startz = complex(0, 0)
        self.zorbit = [startz]
        self.orbit = [self.z_to_xy(startz)]

    def start_orbit(self, x, y):
        self.initialize_orbit()
        self.iter = 1
        self.zc = self.xy_to_z(x, y)

    def update_orbit(self):
        z = self.zorbit[-1]
        zn = (z * z) + self.zc
        if abs(zn) <= 2 and (self.iter < self.itermax):
            self.iter += 1
            self.zorbit.append(zn)
            self.orbit.append(self.z_to_xy(zn))

    def plot_orbit(self, x, y):
        max_iters = 30
        c = self.xy_to_z(x, y)
```

```
        z = c
        saved = empty(max_iters)
        saved[0] = 0
        total = 1
        for i in xrange(1, max_iters):
            z = z*z + c
            distance = abs(z - c)
            saved[i] = distance
            plot(i, distance, 'ro')
            total += 1
            # Check if orbit has diverged
            if abs(z) > 2.0:
                break
        # Plot the points in the orbit
        plot(arange(total), saved[:total])
        xlabel('Iteration')
        ylabel('Distance from c')
        title('Distance of orbit from  c, c = ' + str(c))
        show()

    #------------------------------------------------------------------------
    # Distribution of iterations needed to diverge
    #------------------------------------------------------------------------

    def show_distribution(self):
        image = self.img
        flat = reshape(image, HEIGHT*HEIGHT)
        hist(flat, bins=self.iter + 1, fc='b', ec='b', normed='True')
        av = mean(flat)
        xlabel('Number of iterations')
        ylabel('Relative frequency')
        title('Iterations needed to diverge, mean = ' + str(int(av)))
        show()

    #------------------------------------------------------------------------
    # Feigenbaum diagram for points on a given constant angle from the origin
    #------------------------------------------------------------------------

    def show_feigenbaum(self, x, y):
        max_iters = 1000
        z_pt = self.xy_to_z(x, y)
        theta = angle(z_pt)
        r = linspace(0, abs(z_pt), 1024)
        x_points = r*cos(theta)
        y_points = r*sin(theta)
        c = x_points+complex(0,1)*y_points
        z = copy(c)
        # Initial run to get convergence
        for i in arange(max_iters):
            multiply(z, z, z)
```

```
            add(z, c, z)
            # Check which points have diverged
            rem = abs(z)>2.0
            # z becomes any points which have not yet diverged
            z[rem] = 2.0
        # Plot the orbits
        for t in arange(max_iters):
            plot(r, abs(z), 'k.', markersize=0.1)
            multiply(z, z, z)
            add(z, c, z)
            # Check which points have diverged
            rem = abs(z)>2.0
            # z becomes any points which have not yet diverged
            z[rem] = 2.0
        xlabel('|c|')
        ylabel('Distance from origin of orbit')
        title(r'$z = ' + str(z_pt) + '$')
        show()

    #--------------------------------------------------------------------------
    # Event handlers
    #--------------------------------------------------------------------------

    def mousedown(self, x, y, screen):
        self.current_pos = (x, y)
        if self.mode == MAGNIFY_MODE:
            self.start_magnify(x, y)
        elif self.mode == FEIGENBAUM_MODE:
            self.show_feigenbaum(x, y)
        elif self.mode == ORBIT_MODE:
            self.plot_orbit(x, y)

    def mouseup(self, x, y, screen):
        if self.mode == MAGNIFY_MODE:
            self.end_magnify(x, y)

    def mousemove(self, x, y, screen):
        if self.mode == MAGNIFY_MODE:
            self.move_magnify(x, y)
        elif self.mode == ORBIT_MODE:
            self.start_orbit(x, y)
        self.current_pos = (x, y)

    def keydown(self, key, screen):
        if key == K_a:
            # Switch to angle map
            self.map = ANGLE_MAP
            self.mode = MAGNIFY_MODE
            self.initialize_orbit()
            self.initialize_image()
```

```
        elif key == K_c:
            # Cycle between palettes
            if self.mode == CYCLE_MODE:
                if self.cycle:
                    self.cycle = False
                else:
                    self.mode = MAGNIFY_MODE
            else:
                self.mode = CYCLE_MODE
                self.cycle = True
        elif key == K_d:
            # Switch to divergence map
            self.map = DIVERGENCE_MAP
            self.mode = MAGNIFY_MODE
            self.initialize_orbit()
            self.initialize_image()
        elif key == K_f:
            self.mode = FEIGENBAUM_MODE
        elif key == K_g:
            # Toggle the grid
            self.grid = (self.grid + 1) % 3
        elif key == K_h:
            # Switch to hankel map
            self.map = HANKEL_MAP
            self.mode = MAGNIFY_MODE
            self.initialize_orbit()
            self.initialize_image()
        elif key == K_i:
            # Internal structure - closest approach to origin
            self.map = CONVERGENCE_MAP
            self.mode = MAGNIFY_MODE
            self.initialize_orbit()
            self.initialize_image()
        elif key == K_m:
            # Switch to magnify mode
            self.mode = MAGNIFY_MODE
            self.initialize_orbit()
            self.initialize_image()
        elif key == K_n:
            # Toggle the image normalization
            self.normalize = not self.normalize
        elif key == K_o:
            # orbit mode
            self.mode = ORBIT_MODE
            self.initialize_orbit()
        elif key == K_p:
            # Orbit period - which period is the orbit closest to
            self.map = PERIOD_MAP
            self.mode = MAGNIFY_MODE
            self.initialize_orbit()
```

```
            self.initialize_image()
        elif key == K_r:
            # restart in magnify mode
            self.mode = MAGNIFY_MODE
            self.initialize_area()
            self.initialize_orbit()
            self.initialize_image()
        elif key == K_s:
            # Show the distibution of iterations needed to diverge
            self.show_distribution()
        elif key == K_t:
            # Cycle map - period of the orbit
            self.map = CYCLE_MAP
            self.mode = MAGNIFY_MODE
            self.initialize_orbit()
            self.initialize_image()
        elif key == K_SPACE:
            # Turn pause on or off
            self.paused = not self.paused
        elif key == K_UP:
            # Halve the spacing shown between levels
            self.levels *= 2
        elif key == K_DOWN:
            # Double the spacing shown between levels
            if self.levels > 1:
                self.levels /= 2
        elif key in NUMBER_KEYS:
            # Change the palette
            palettes = self.palettes
            num_palettes = len(palettes)
            key_num = NUMBER_KEYS.index(key)
            # Check we're not indexing past the end of the palettes
            if key_num < num_palettes:
                new_palette = palettes[key_num]
                self.palette = new_palette
                screen.set_palette(new_palette)

#---------------------------------------------------------------------------
#
# Pygame Initialization
#
#---------------------------------------------------------------------------

pygame.init()
screen = pygame.display.set_mode(SCREEN_SIZE, 0, 8)
main = Mandelbrot()
main.render(screen)

pygame.display.set_caption("Mandelbrot Explorer by Adam Cunningham")
main.add_palette(screen.get_palette())
```

```
pygame.display.update()

#------------------------------------------------------------------------
#
# Main event handling loop
#
#------------------------------------------------------------------------

on = True

while on:

    for event in pygame.event.get():
        if event.type == QUIT:
            on = False
            pygame.quit()

        if event.type == KEYDOWN:
            main.keydown(event.key, screen)

        if event.type == MOUSEBUTTONDOWN:
            x, y = pygame.mouse.get_pos()
            main.mousedown(x, y, screen)

        if event.type == MOUSEBUTTONUP:
            x, y = pygame.mouse.get_pos()
            main.mouseup(x, y, screen)

        if event.type == MOUSEMOTION:
            x, y = pygame.mouse.get_pos()
            main.mousemove(x, y, screen)

    main.update()
    main.render(screen)

    pygame.display.flip()
```