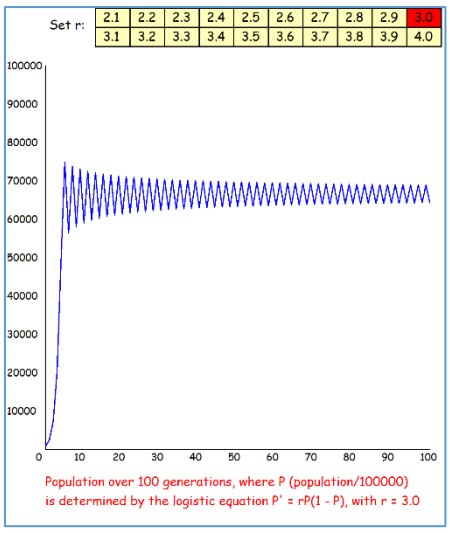
Turtle Python 7 – Chaotic Phenomena

Until the 1970s, scientists in many fields focused their attention almost entirely on *linear systems* – that is, systems governed by *linear* equations (of the general form $y = ax + b$). Such systems are "well-behaved" and relatively easy to analyse, because small changes in "inputs" lead to correspondingly small changes in "outputs", so approximation methods work well. But as fast-improving computer technology enabled real physical systems to be modelled in more complex ways, users began to realise that these newer models were not at all well-behaved in this way. Most famously, in 1961, Edward Lorenz – working with a computer model of the weather – discovered that a tiny difference in an initial value entered into the model to represent the condition at one point in time (a value of 0.506, used as an approximation for 0.506127) could lead to totally different future predictions. He thus encountered *sensitive dependence on initial conditions*, which he named "the butterfly effect", based on the idea that, if the weather is correctly modelled by equations like those he was working with, then in principle the disturbance caused by the flapping of a butterfly's wings in Brazil could make the crucial difference that later brings about a tornado in Texas.

Note that this sort of behaviour, though commonly called "chaotic" and extremely hard to predict in detail, is not *random*. Indeed we shall see that it can arise in systems that are entirely *deterministic*, where everything follows according to precisely defined rules, with no role whatever for *chance* or genuine indeterminacy. And it turns out that *non-linear systems*, commonly exhibiting this sort of sensitive dependence, occur throughout nature, so that the linear systems deeply investigated by conventional science (and therefore generally assumed in the past to be typical) are the exception rather than the rule. Indeed the mathematician Stanislaw Ulam famously remarked that considering "non-linear systems" as a special subject of study is rather like considering most of biology as the study of "non-elephants"! In recent years, scientific attention has increasingly turned towards these more complex systems, and it is the development of the computer which has made this possible (as illustrated by how the phenomenon of chaos was widely ignored before the 1960s, even though Henri Poincaré had identified it as early as 1890).

1. The Logistic Equation



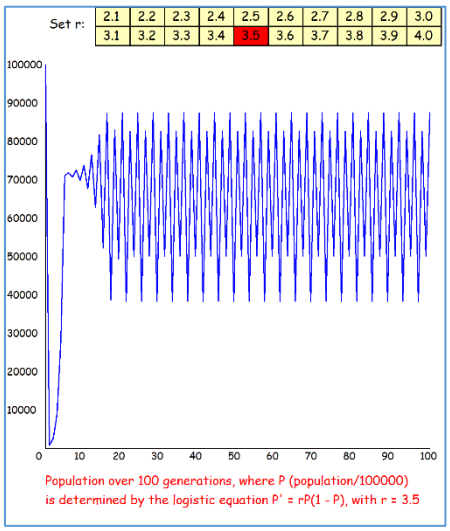
Suppose we have a population of creatures – insects, for example – whose generations do not overlap. Adults lay eggs, and by the time the eggs hatch, the parents have died. If resources of food and space are unlimited, then more adults will typically produce more surviving offspring: for example, the number of offspring that survive long enough to become breeding adults in their turn might be 3 times the number of adults. This is a straightforward *linear* relationship. But as Thomas Malthus famously pointed out in his *Essay on the Principle of Population* of 1798 (a work which Darwin credited for inspiring his discovery of evolution by natural selection), such multiplicative population growth quickly becomes unsustainable. If we start with 30 insects weighing 5 mg each, and the population multiplies by 3 every year, then after 60 years, the weight of the insect population will exceed that of the entire Earth! So any environment will impose some absolute limit on population growth, and the environmental pressure will typically increase as the population grows closer to that limit. Suppose, for example, that our insects live mainly on a particular kind of plant; then as the population of insects grows, those plants are likely to be damaged and their population seriously reduced from being eaten, and hence only a small proportion of the next generation of insects will be able to find the food to grow successfully to maturity. (Similar issues are familiar from the growth of human populations, where only the increasing use of technology has – at least so far – prevented devastation of our own numbers by environmental pressure.)

The dynamics of these sorts of interactions are likely to be complicated in detail, but it turns out that a wide range of models exhibit broadly similar behaviour, so we can confine our attention here to an extremely simple model, based on what is commonly called the *logistic equation*:

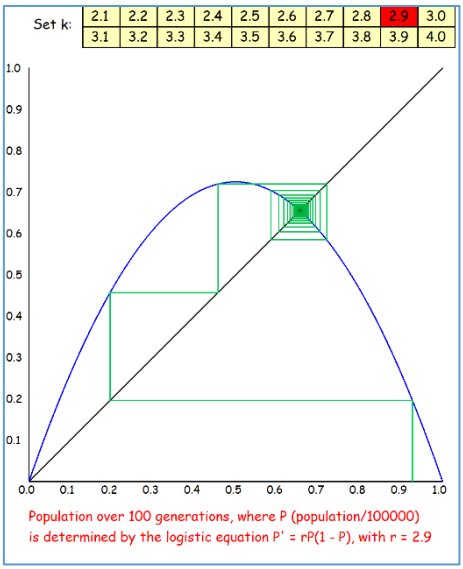
$$P' = rP(1 - P)$$

Here *P* is the current population of insects, as a proportion of the limit that can ever be achievable within that environment (so if the limit is 100000 and the current population is 27183, then *P* is 0.27183). The parameter *r* is the *growth rate*, and the equation predicts *P'*, the population in the next generation, again as a proportion of the limit. In the previous paragraph, we imagined a growth rate of 3, generating an exponential population explosion as it multiplied, generation after generation. But in this logistic equation, any such explosion is prevented by the (1 – *P*) factor, which approaches 0 as the population comes close to the limit (and if *P* actually reaches the limit of 1, then *P'* will plunge to zero, eradicating the population).

What result should we expect from all this? Prior to the non-linear revolution, most scientists would probably have proceeded by analysing the conditions for *equilibrium* (a method still standard in economics). This involves assuming that the population will eventually stabilise, at which point *P'* will be identical to *P*, and for this to happen with *r* equal to 3, (1 – *P*) must be 1/3 so that *P* is 2/3. On this basis, we might predict that if the limiting population is 100000, then almost no matter what population number we start from, we should end up with an equilibrium at 66667 (or, since rounding is involved, a minimal oscillation between 66667 and 66666). But in fact, as the image of the "Logistic" program illustrates above, that very rarely happens (and only if we happen to start from a lucky value). Instead, we nearly always quickly reach a population that oscillates around that equilibrium, but without ever subsequently getting closer to it. If you click repeatedly on the red "3.0" on the Canvas, then the program will be run repeatedly with a random initial population, and you can see this for yourself.



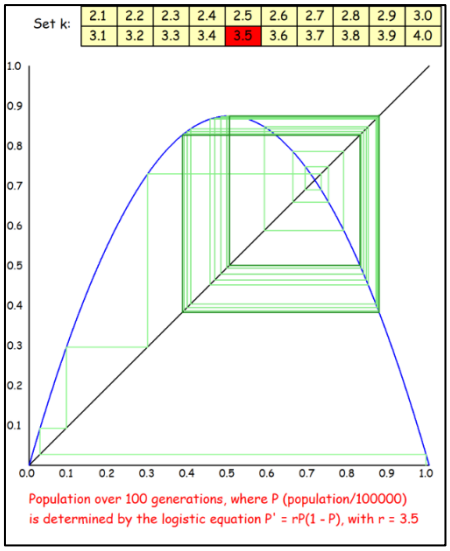
If *r* is less than 3.0, then in fact an equilibrium does usually get reached, but things are more interesting if we select a larger value for *r*, by clicking on "3.5", for instance. Again equilibrium is theoretically possible (at a rounded value of 71429), but now the population will almost always end up oscillating through four different values (50091, 87500, 38281, and 82693). Take *r* higher still, to 3.6, 3.7, 3.8. 3.9 or 4.0, and the population movements become even more surprising, with increasing apparent randomness or – as it is now known – *chaotic* behaviour. (An ambitious student might be interested in modifying the program to allow finer discrimination, e.g. selectable *r* values between 3.51 and 3.59. Theoretically, chaos ensues at 3.57.)

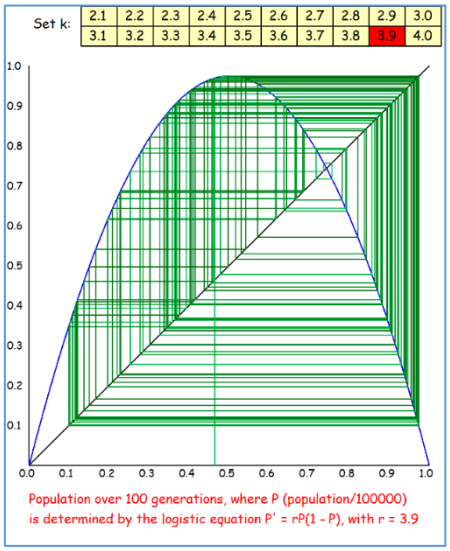


What causes this very surprising behaviour, from such a simple equation? A useful way of visualising what is going on is a "spider diagram", implemented by the program "Logistic Spider" (and shown in the image to the left). Both the x- and y-axes here run from 0.0 to 1.0, and the line $y = x$ is also shown (making a 45° angle with both axes). Against this background is plotted the graph of the logistic function $y = rx(1 - x)$, in blue, with *r* taking the chosen value. In the plot shown here, *r* is 2.9 and the initial population has been randomly given the value of 92701 – so we start with *x* equal to 0.92701 on the graph. If we now put this value into the logistic function, *y* becomes 0.19622 (the height of the green line going up from the x-axis just above 0.9). So according to this model, the population has fallen dramatically, from 92701 down to 19622. Where next? The height we have

reached on the graph (i.e. 0.19622) now has to be fed in as the next x-value, so to do this, we draw a horizontal line to hit the line $y = x$; this is the fairly long horizontal green line at around height 0.2, whose left-hand end is at the point (0.19622, 0.19622). Now if we move vertically until we reach the blue curve of the logistic function, the position we reach will correspond to the y-value we get from putting x equal to 0.19622, namely 0.45738 (which corresponds to a population of 45738). To get the next point in the series, we again move horizontally until we hit the line $y = x$, now at the point (0.45738, 0.45738), and then move vertically to hit the blue curve, so as to find the y-value that corresponds to the x-value 0.45738 (namely, $y = 0.71973$). Notice how repeatedly moving to the line $y = x$ serves to translate each successive y-value into an x-value for the next step. And you will see from the diagram how the series converges onto the point of intersection of the blue curve and the line $y = x$: equilibrium is reached at (0.65517, 0.65517).

If r is given the value 3.5 instead of 2.9, then the shape of the intersection between the logistic function and the line $y = x$ crucially changes, with the result that "near misses" are now pushed further away from the equilibrium (at 0.714285) by subsequent iterations, as we see pictured here. The sequence gets close with the fourth value of 0.72974, but then diverges (with 0.69027, 0.74829, 0.65923, 0.78626, 0.58819, 0.84778, and 0.45167) before converging, after 16 intermediate iterations, to the repeating four-value cycle 0.87500, 0.38281, 0.82693, and 0.50091. Almost any initial value (apart from a "bullseye" on the equilibrium) will likewise converge to this cycle, which we therefore call the *attractor* of the function. When r was 2.9, we saw that the attractor of the logistic function was a single equilibrium value; when r is 3.5, the attractor becomes a four-value cycle; what happens if we raise r yet further?





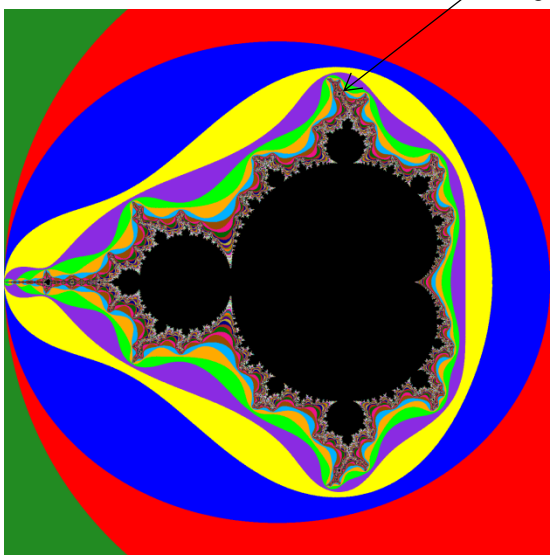
If r is given the value 3.9, then as with 3.5, any "near miss" of the equilibrium point (which is at 0.74359) will be followed by iterations that get further away, but now there is no settling down into a simple cycle of values. Instead, as shown here, the sequence quickly gets into a *chaotic* trajectory, jumping around unpredictably and exhibiting a *sensitive dependence* in which nearby values are followed by diverging rather than converging patterns. This erratic trajectory is an *attractor* – in that almost any initial value will be "pulled" into something like it – but it is a "strange attractor", because it is divergent rather than convergent. The population will never reach stability, but will be repeatedly subject to "booms" and "busts", driven into these not by any external factors, but simply by the non-linear dynamics of the logistic equation.

Lord Robert May brought the logistic equation to prominence in a famous 1976 article in the journal *Nature*, whose conclusion focuses on its *educational* significance. There May points out that school and university courses have hitherto been dominated by "the elegant body of mathematical theory pertaining to linear systems", thus developing students' "mathematical intuition" in a way that "ill equips [them] to confront the bizarre behaviour exhibited by … nonlinear systems [which] are surely the rule, not the exception, outside the physical sciences". He finishes as follows: "***I would therefore urge that people be introduced to [the logistic equation] early in their mathematical education. … Such study would greatly enrich the student's intuition about nonlinear systems. Not only in research, but also in the everyday world of politics and economics, we would all be better off if more people realised that simple nonlinear systems do not necessarily possess simple dynamical properties.***"

2. The Mandelbrot Set

With the logistic equation, we saw how a sequence of numbers could be generated in turn, by starting with one given number, feeding it into the equation (as *P*) to derive the next number in the sequence (as *P'*), then feeding this number in to get the next, and so on. We also saw that if *r* takes a relatively high value (e.g. 3.6 or above), then the sequence is *chaotic*, so that it becomes impossible to predict any simple long-run outcome. This also yields *sensitive dependence on initial conditions*, in which even tiny differences at one point in the sequence will magnify as the following numbers are generated, making it impossible to treat one "trajectory" through the numbers as a good approximation of other "nearby" trajectories – any such nearness is purely temporary.

See §2.1



An even more famous illustration of these ideas is provided by the *Mandelbrot set*, named after Benoit Mandelbrot (1924-2010) and pictured here as the black area. This is of great mathematical interest, but here we focus on how the image is constructed. The canvas is 1500×1500 pixels, and it here represents – at a *scale* value of 500 pixels per unit – the 3.0×3.0 square in the *complex plane* from –2.0 to 1.0 horizontally, and –1.5 to 1.5 vertically. Thus each point represents a complex number, of which the general form is ($a + ib$), where i is the square root of –1.

(If you are unfamiliar with complex numbers, just consider instead starting with the point (a, b) using standard Cartesian coordinates – the sort everyone learns at school – and following a sequence of points on the graph.)

Now suppose we pick a particular complex number, say $z_0 = (a + ib)$, and form a sequence starting from this, using the formula:

$$z' = z^2 + (a + ib)$$

In other words, if z is some number in the sequence, then we calculate z', the next number in the sequence, by squaring z and adding ($a + ib$), which is z_0, the first point in the sequence. Using "z_n" to signify the number reached after n iterations of this process, we can express this as:

$$z_{n+1} = {z_n}^2 + z_0$$

Squaring the complex number ($x + iy$) can be done by simple algebra, bearing in mind that i is the square root of –1 (so that $i^2y^2 = -y^2$):

$$(x + iy)^2 = x^2 + i^2y^2 + 2ixy = (x^2 - y^2) + i(2xy)$$

Thus squaring the number represented by the coordinates (x, y) takes us to the number represented by the coordinates ($x^2 - y^2$, $2xy$), and if we then add ($a + ib$), this takes us to ($x^2 - y^2 + a$, $2xy + b$). Note again that this can be understood as generating a sequence of points in the graph, so thinking in terms of complex numbers is not essential.

In updating the x- and y-coordinates *within our image* from one point to the next in this series, we need to bear in mind the *scale* value of 500 pixels per unit. Thus a *numerical* x-coordinate of m will have a *pixel* x-coordinate of 500m, a *numerical* y-coordinate of n will have a *pixel* y-coordinate of 500n, and a *numerical* y-coordinate of $m \times n$ should have a *pixel* y-coordinate of 500mn (=500m × 500n ÷ 500). Notice therefore that when we multiply two of these pixel coordinates together, we need to divide by the *scale* value (here 500, but changeable if we want either more detail or more speed). So, for example, if we have pixel coordinate values of x as 700 and y as 1000, then the pixel coordinate value of xy should come to 1400 (i.e. 700×1000/500), as would be expected given that the y-value of 1000 represents the number 2.0. Taking

this into account, you might very reasonably expect the updating code to be something like this:

```
temp = (x*x-y*y) / scale  # temp = x squared minus y squared
y = 2*x*y/scale + b       # calculate new y as (2xy + b)
x = temp + a              # calculate new x as (temp + a)
```

Here the variable "temp" is used to store the current *scale-adjusted* value of $x^2 - y^2$, prior to *y* being updated. Then *y* is updated to $2xy + b$ (again adjusting *xy* by the *scale* factor), and *x* is updated to $x^2 - y^2 + a$.

In the "Mandelbrot" example program, however, you will see that the updating is done slightly differently, like this:

```
temp = divmult(x+y, scale, x-y);
y = divmult(2*x, scale, y) + b;
x = temp+a;
```

Any computer system will have a limit to the size of integers that it can handle, and because *Turtle* uses 32-bit (i.e. 4 byte) integers, its highest positive integer is $(2^{31}-1)$, i.e. 2147483647 (which is accordingly named *maxint*). So if you want to produce images of parts of the Mandelbrot set that involve a high *scale* value (by "zooming in" on particular parts of it, as in §2.1 below), then squaring *x* or *y* directly will probably exceed this limit and generate an error message. *Turtle* functions such as *hypot* and *divmult* are designed to circumvent this sort of problem, with *hypot*(*a*,*b*,*c*) yielding the rounded result of $c\times\sqrt{(a^2+b^2)}$,[1] and *divmult*(*a*,*b*,*c*) yielding the rounded result of $c\times(a/b)$, while avoiding overflow errors for intermediate results. In the current case, we first want to calculate $(x^2-y^2)/scale$, so we take advantage of the well-known formula for *difference of two squares*: $(x^2-y^2) = (x+y).(x-y)$; hence we use `divmult(x+y,scale,x-y)`. Then we calculate $2xy/scale$ using `divmult(2*x,scale,y)` so that the multiplication of *x* by *y* takes place safely within the function, with division by *scale* bringing the function's result down below *maxint*.

The Mandelbrot set is the set of points (*a*, *b*) in the complex plane for which the sequence just explained – starting from (*a*, *b*) – never *diverges* in the sense of getting further and further from the origin at (0, 0). Any sequence that moves further than 2.0 from the origin will diverge, and in order to produce our brightly coloured diagram, we need to keep track of *how quickly* sequences exceed this crucial distance, so our program combines these tests with the updating calculation as follows:

```
iterations = 0;
while (hypot(x,y,1) < 2*scale) and (iterations <= maxcol):
    temp = divmult(x+y,scale,x-y)
    y = divmult(2*x,scale,y)+b
    x = temp+a
    iterations += 1  # this adds 1 to iterations
```

The loop terminates when *either* `hypot(x,y,1)` – the Pythagorean distance of (x, y) from the origin – exceeds 1000 (double the *scale* value, hence equivalent to a distance of 2.0 in the complex plane), *or* the number of iterations exceeds *maxcol*, a constant set to 40 at the beginning of the main program. So if a sequence has not diverged sufficiently after 40 iterations, then it is presumed never to diverge (which is not strictly correct, but enables us to limit the time that the program will take); hence its starting point is counted as part of the Mandelbrot set and accordingly coloured black. But if a sequence does diverge within 40 iterations, then the starting point is coloured with the *Turtle* colour code corresponding to the *iterations* value (between 1 and 40 inclusive) at which the loop terminated. Thus points which diverge after 1 iteration are *green*, after 2 iterations *red*, then *blue*, *yellow*, *violet* and so on; this is how we get the brightly coloured image shown earlier.

---

[1] The name *hypot* derives from *hypotenuse*, because *hypot*(*a*,*b*,*c*) calculates the hypotenuse of a right-angled triangle with sides *a* and *b*, multiplied by *c*. So, for example, *hypot*(1,2,1000000) gives 2236068, which is the square root of 5 multiplied by a million and rounded. To print out $\sqrt{5}$ to six places of decimals, use "print(qstr(2236068,1000000,6))" – the function *qstr*(*a*,*b*,*c*) generates the decimal string of *a*/*b*, rounded to *c* places after the decimal point.

2.1 Zooming in on the Mandelbrot Set

The "Mandelbrot" example program starts with a subroutine called "startprompt", which offers the user two prompts that allow a choice of program options. The first is as follows, offering either to display the whole Mandelbrot set, or alternatively to zoom in on the "mini lake" indicated in the last image:

> Select Whole set, or Zoom on mini "lake" at –0.1592, –1.0330 (W/Z)

- If "W" is pressed, then "xcentre" is made equal to –500000, "ycentre" is made equal to 0, and the second prompt is:

  > Select Fast/Medium/Slow, giving resolution 300/750/1500: (F/M/S)

  Now pressing "F", "M" or "S" will result in "scale" values of 100, 250, and 500 respectively, with "pixels" equal to 300, 750 and 1500 (i.e. scale*3).

- If "Z" is pressed in response to the first prompt, then "xcentre" is made equal to –159200, "ycentre" is made equal to –1033000, and the second prompt is:

  > Select Fast/Medium/Slow, giving resolution 300/600/1200: (F/M/S)

  Now pressing "F", "M" or "S" will result in "scale" values of 10000, 20000, and 40000 respectively, with "pixels" equal to 300, 600 and 1200 (i.e. scale/100*3).

Note that *xcentre* and *ycentre* here represent *real number coordinates divided by one million*, which for the "W" option specify the centre point (–0.5, 0) and for the "Z" option the centre point (–0.1592, –1.033).

Having returned from this subroutine, the main program continues to calculate a series of *pixel coordinates* (i.e. coordinates in terms of position on the Canvas):

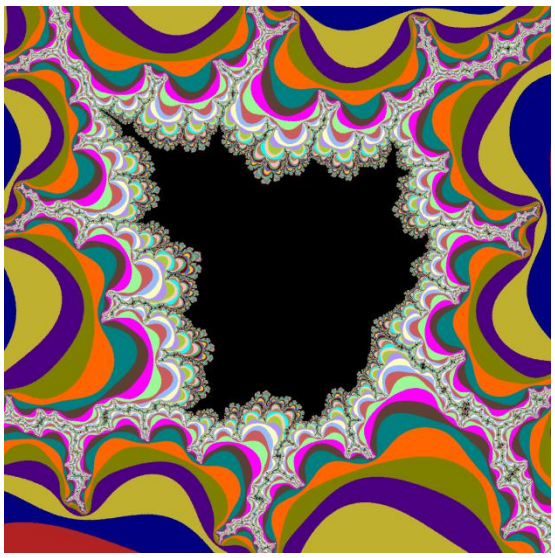
```
xstart=divmult(xcentre,1000000,scale)-pixels/2
ystart=divmult(ycentre,1000000,scale)-pixels/2
xfinish=xstart+pixels
yfinish=ystart+pixels
```

The first two lines calculate the *pixel* coordinate of the relevant centre point, then subtract half the Canvas pixel width (or height) to yield the pixel coordinates of the point (*xstart, ystart*), which is the top left of the Canvas. Then the next two lines calculate the pixel coordinates of the point (*xfinish*, *yfinish*), which is the bottom right of the Canvas. Here are the relevant values for the different startprompt choices:

	W/F	W/M	W/S	Z/F	Z/M	Z/S
scale	100	250	500	10000	20000	40000
pixels	300	750	1500	300	600	1200
xstart	–200	–500	–1000	–1742	–3484	–6968
xfinish	100	250	500	–1442	–2884	–5768
ystart	–150	–375	–750	–10480	–20960	–41920
yfinish	150	375	750	–10180	–20360	–40720

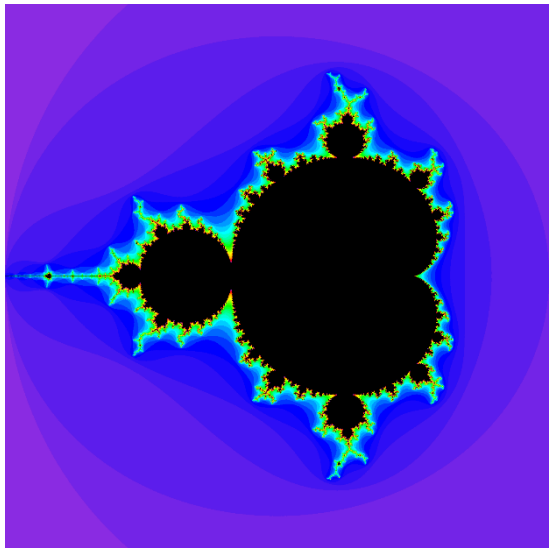
The program then displays the range over which the Mandelbrot set will be plotted, from *xstart* to *xfinish* horizontally, and *ystart* to *yfinish* vertically, in every case divided by *scale* to convert to a real number (and displayed here to four decimal places). To sum up, then, *pixels* determines the size of the Canvas (a square measuring *pixels*×*pixels*), and *scale* determines the magnification of the image relative to the complex plane – hence the extent of the complex plane that is pictured will be determined by the *pixels*/*scale* ratio. The positioning of this image in the complex plane is determined by *xcentre* and *ycentre*, specifying the mid-

point of the image in units of 0.000001 (i.e. millionths). The "Z" option selects (–0.1592, –1.033) as the mid-point in the complex plane, which as noted earlier is close to the centre of the "mini lake" indicated in the previous image (but of course you can choose other locations to explore if you wish). Here we see the zoomed version – and notice how when we zoom in, we still see lots of fine detail, which indeed goes "all the way down". This therefore exhibits the sensitive dependence on initial conditions mentioned earlier – even points which look adjacent at one magnification can behave quite differently if we look more closely (but unfortunately the limited precision of the *Turtle System*'s number variables – four bytes and therefore 32 binary digits – prevents us from pushing the magnification much further than this).



2.2 Spectral Colouring

So far, our colouring of the Mandelbrot set has used the built-in sequence of *Turtle* colours, which sets adjacent colours as contrasting rather than blending. These images are striking, but it is more illuminating to use colours that change gradually, to indicate more consistently the rate of divergence at different points in the complex plane. A neat way of doing this is illustrated in the "Mandelbrot spectral colours" example program, whose output is pictured here. To achieve this, we set up a "spectrum" of reference colours starting with *violet*, then *blue*, *cyan*, *lime*, *yellow*, *orange*, *red*, and back to *violet* again (these colour codes are stored in the *spectcol* list, so *spectcol*[0] = *violet*, *spectcol*[1] = *blue*, and so on). Then we calculate "boundaries" for each of these (stored in the *boundary* list), spread through the range from 0 to *maxcol*, so that if *maxcol* is 40, their successive boundary values are: *violet* 0, *blue* 6, *cyan* 11, *lime* 17, *yellow* 23, *orange* 29, *red* 34, and again *violet* 40; whereas if *maxcol* is 100, their successive boundary values are: *violet* 0, *blue* 14, *cyan* 29, *lime* 43, *yellow* 57, *orange* 71, *red* 86, and again *violet* 100. Then if we want a colour corresponding to number 32, say, we identify the two boundaries on either side of this (i.e. if *maxcols* is 100, these will be *cyan* at 29 and *lime* at 43), and we mix the corresponding colours in proportion to our number's closeness to the relevant boundaries (so for number 32, which is 3 away from the *cyan* boundary and 11 away from the *lime* boundary, we mix 3/14 *lime* with 11/14 *cyan*). Here is the function that does the mixing:



```
def mixcolour(n):
    col2 = 1
    while (boundary[col2]<n) and (col2<7):
        col2 += 1
    col1 = col2-1
    return mixcols(spectcol[col1], spectcol[col2], boundary[col2]-n,
                   n-boundary[col1])
```

The parameter *n* is the number (e.g. 32) whose colour we want to find, and we do this by identifying the boundaries (*col1* and *col2*) between which it lies. The result is conveniently provided for us using *Turtle*'s "mixcols" command, which mixes the two given colours in the given proportions (by calculating a weighted average of each of the red, green and blue components of each colour, then combining these).

3. Ideas for Independent Exploration: Mandelbrot and Julia Tourism

This handout has been mainly about mathematical understanding rather than program development, but there is plenty of scope for independent exploration. Here are two initial ideas:

- As mentioned in §1, many functions display similar chaotic behaviour to the *logistic* equation, so you could try discovering other such interesting functions and graphing them in a similar way (making sure that they map values from the relevant range – e.g. 0.0 to 1.0, or 0 to 1000 – into that same range). Other famous examples that you might want to look up are the *Henon map*, *sine map*, and *tent map*.
- In §2.1 and §2.2, we saw a program that "zooms in" on the Mandelbrot set. Try viewing some well-known "tourist sights", such as "Sea Horse Valley" (centre around (-0.79,0.15), e.g. with a scale of 5000 and 1000x1000 pixels) or "Elephant Valley" (centre around (0.34,0.127), e.g. with a scale of 5000 and 1300x1300 pixels). If you're feeling seriously ambitious, try to develop a Mandelbrot program that works more efficiently (e.g. by spending less time scanning large black expanses), or which enables you to specify a "zoom area" with the mouse.

We can also dig more deeply into the mathematical theory, to uncover further possibilities for generating beautiful patterns. Closely related to the Mandelbrot set are so-called *Julia sets*, named after the French mathematician Gaston Julia (1893-1978), who taught Benoit Mandelbrot at the École Polytechnique in Paris from 1945-47 (Mandelbrot's family having emigrated from Poland to France in 1936). We saw in §2 that the Mandelbrot set is the set of complex numbers ($a + ib$), such that a series of the form:

$$z_0 = (a + ib)$$
$$z_{n+1} = {z_n}^2 + (a + ib)$$

does not diverge to infinity (or equivalently, never gets further than 2.0 from the origin). To generate the Mandelbrot image, we visit in turn lots of points ($a + ib$) that lie within the circle of radius 2.0 around the origin, then iteratively calculate values of this series for each of them, and then assign a colour showing how quickly the series diverges (up to some suitable maximum number of iterations, e.g. 40).

Julia sets are subtly different, and each one of these is relative to a specific complex *parameter* – so there are zillions of Julia sets, each corresponding to a different parameter value. The Julia set for any such chosen parameter ($p + iq$) is the set of complex numbers ($a + ib$), such that a series of the form:

$$z_0 = (a + ib)$$
$$z_{n+1} = {z_n}^2 + (p + iq)$$

does not diverge to infinity (or equivalently, never gets further than 2.0 from the origin). The process of generating the Julia image (for any chosen parameter value) is accordingly much the same as with the Mandelbrot set: we visit in turn lots of points ($a + ib$) that lie within the circle of radius 2.0 around the origin, then iteratively calculate values of this series for each of them, and then assign a colour showing how quickly the series diverges (up to some suitable maximum number of iterations, e.g. 40).

- As with the Mandelbrot set, you can do "Julia tourism" by exploring various Julia sets that have impressed some people enough to have been given shape-based nicknames, for example involving parameter values of –1.755 (Airplane), –1.0 (Basilica), –0.123+0.745*i* (Douady's Rabbit), –0.8+0.156*i* (Dragon), –0.1+0.651 (Lightning), –0.5+0.567*i* (Phoenix), –0.75 (San Marco), and –0.745+0.113*i* (Twin Spirals). Note that some of the features named here might only be visible when magnified, and only in particular regions of the set. It's also worth exploring around parameters that are at interesting places within the Mandelbrot set (e.g. Seahorse Valley). Here it might be helpful to take a preliminary look at the Julia sets using the convenient app at https://sciencedemos.org.uk/julia.php. The companion app https://sciencedemos.org.uk/mandelbrot.php is also instructive – for instance, you will find that the pattern can change quite significantly as you adjust the number of iterations (e.g. from 40 up to 400).