Lesson 14: Domain of attraction

```
> restart;
  with(VectorCalculus,Jacobian):
  with(plots):
```

How close is "close enough"?

If you start Newton's method at a point close enough to a solution, it should converge rapidly to that solution. But how close? We were looking at a two-variable example.

```
> F:= X -> <X[1]^2 * X[2] + X[1]*X[2]^2 - 2,
  X[1] + sin(X[2]) - 2>;
```

$$F := X \to \langle X_1^2 X_2 + X_1 X_2^2 - 2, X_1 + \sin(X_2) - 2 \rangle \quad (1.1)$$

```
> J:= unapply(Jacobian(F(X),[X[1],X[2]]),X);
  Newt:= X -> evalf(X - J(X)^(-1) . F(X));
```

$$J := X \to rtable\Big(1..2, 1..2, \{(1,1) = 2 X_1 X_2 + X_2^2, (1,2) = X_1^2 + 2 X_1 X_2, (2,1) = 1, (2,2) = \cos(X_2)\}, datatype = anything, subtype = Matrix, storage = rectangular, order = Fortran_order\Big)$$

$$Newt := X \to evalf\left(X - \left(\frac{1}{J(X)}\right).F(X)\right) \quad (1.2)$$

We found two solutions that I'll call Sol1 and Sol2.

```
> Sol1:= <1.33623379100000,0.725843022900000>;
  Sol2:= <2.41188234700000,-2.71707384800000>;
```

$$Sol1 := \begin{bmatrix} 1.33623379100000 \\ 0.725843022900000 \end{bmatrix}$$

$$Sol2 := \begin{bmatrix} 2.41188234700000 \\ -2.71707384800000 \end{bmatrix} \quad (1.3)$$

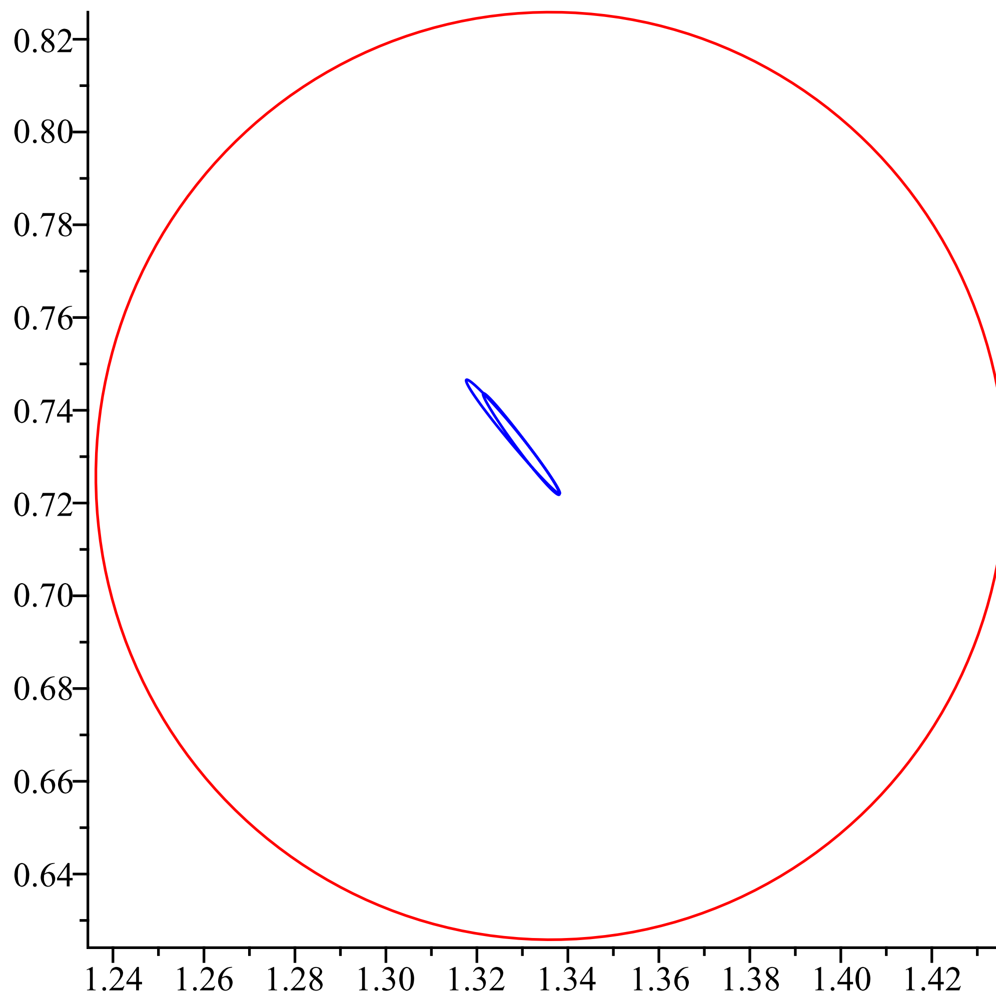
To investigate our question, I wanted to plot circles of various radii centred at the solution together with their images under **Newt**.

Suppose we find that for every radius r with $0 < r \le R$, the image of the circle is always inside a slightly smaller circle. Then, starting from any point whose distance from the solution is less than R, you will always get convergence to the solution.

We plotted two parametric curves. One (in red) is the circle of radius R = 0.1 centred at the solution we found, the point Sol1, and the other (in blue) is the image of that under **Newt**.

```
> R:= 0.1:
  plot([[Sol1[1] + R*cos(t), Sol1[2] + R*sin(t),
```

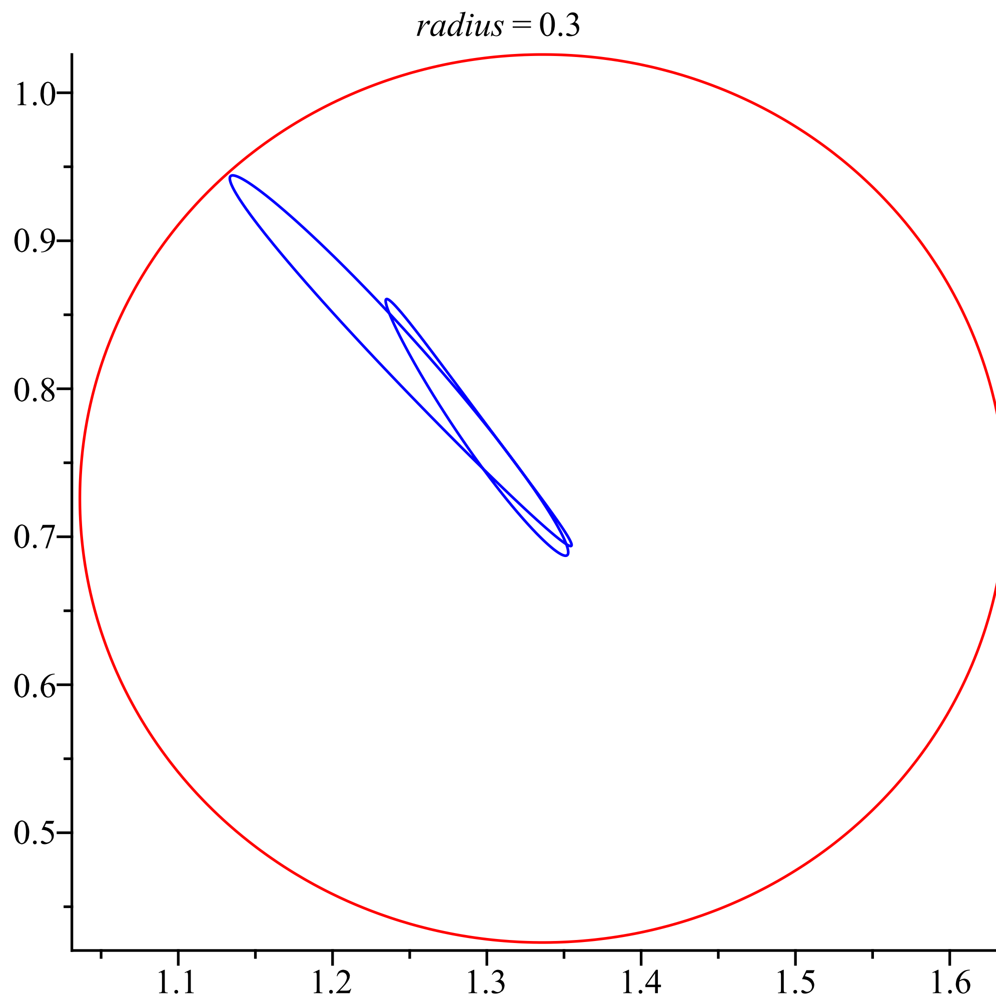
```
    t = 0 .. 2*Pi],
  [ Newt(Sol1+<R*cos(t),R*sin(t)>)[1],
    Newt(Sol1+<R*cos(t),R*sin(t)>)[2],
    t = 0 .. 2*Pi]], colour=[red,blue]);
```



The blue curve is the image of the red circle under Newt, and it's well within it.

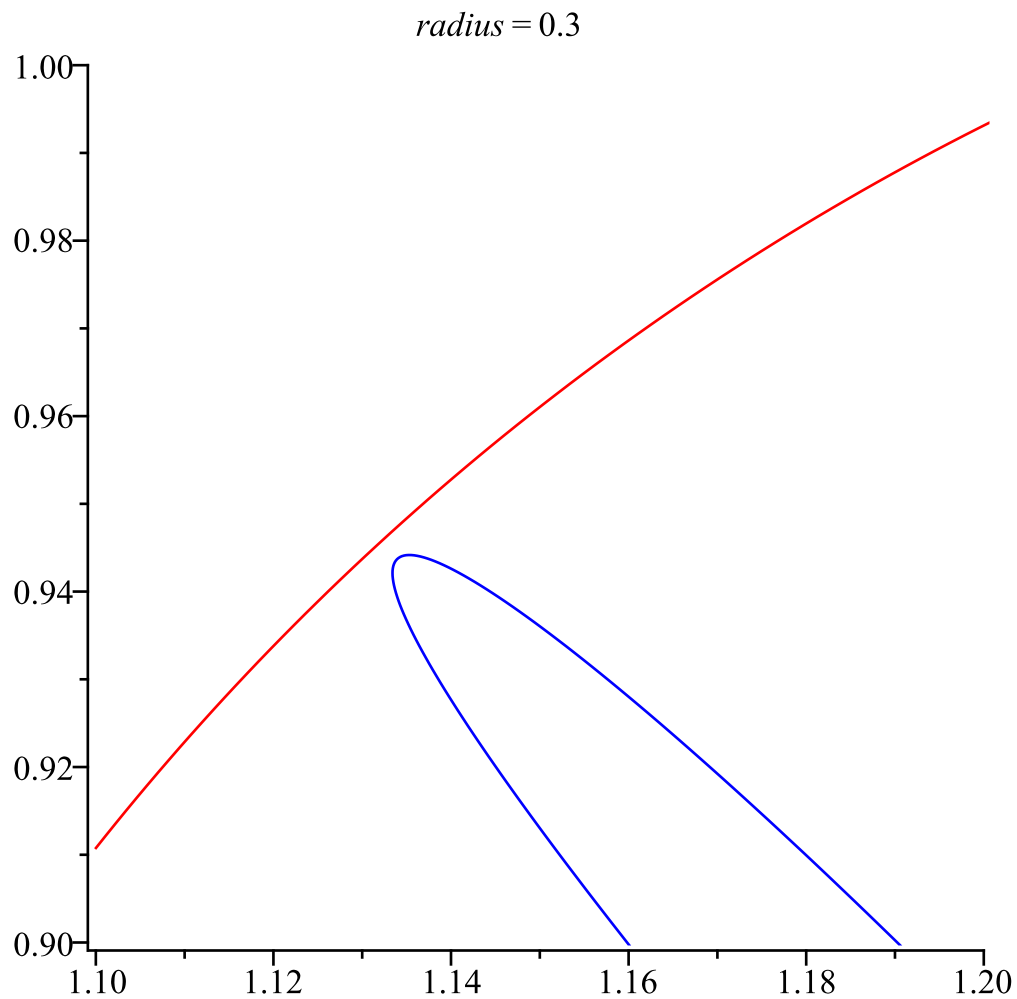
Let's try it for other values of R. It's probably a good idea to put a title on our plots saying which value of R we're using.

```
> R:= 0.3:
  plot([[Sol1[1] + R*cos(t), Sol1[2] + R*sin(t),
    t = 0 .. 2*Pi],
  [ Newt(Sol1+<R*cos(t),R*sin(t)>)[1],
    Newt(Sol1+<R*cos(t),R*sin(t)>)[2],
    t = 0 .. 2*Pi]], colour=[red,blue], numpoints=1000,title =
  (radius=R));
  P3:= %:
```



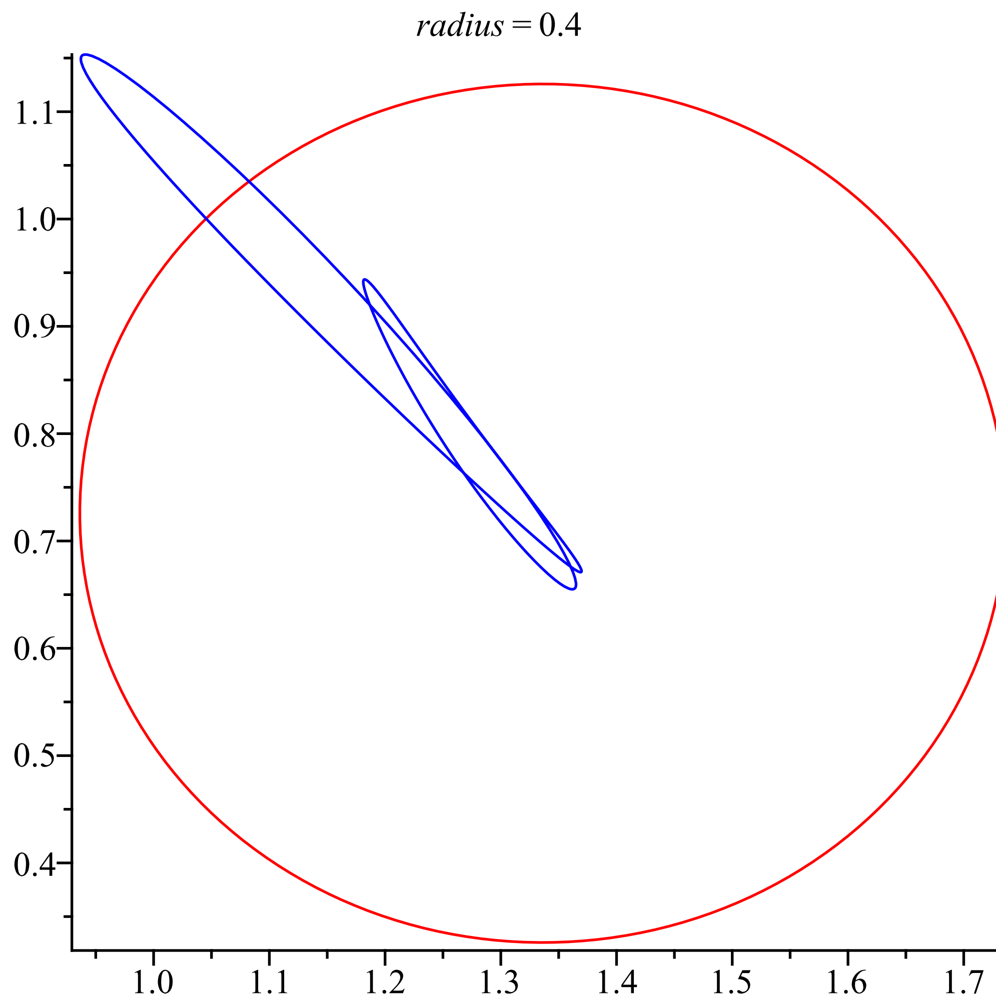
That's just about a borderline case. Can we take a closer look?

```
> display(%,view=[1.1 .. 1.2, 0.9 .. 1]);
```



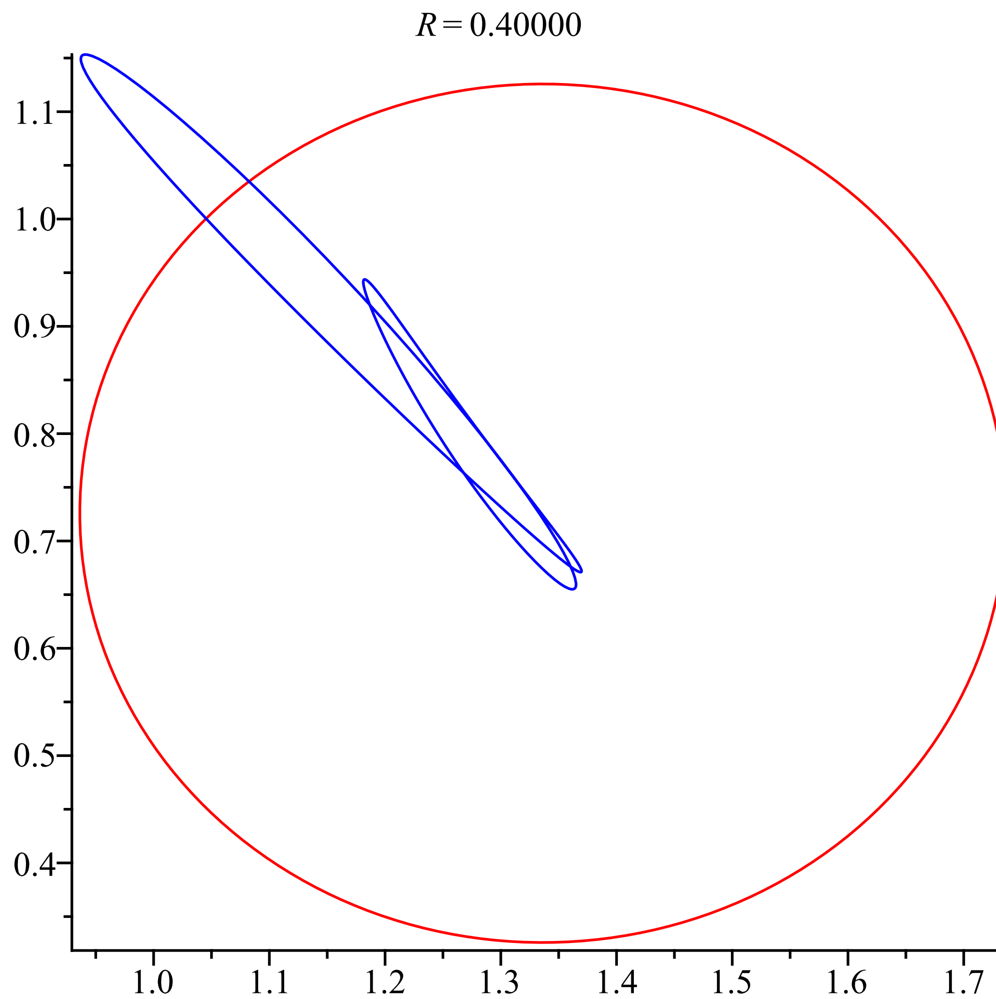
Yes, the blue curve is just barely inside the red circle.

```
> R:= 0.4:
  plot([[Sol1[1] + R*cos(t), Sol1[2] + R*sin(t),
    t = 0 .. 2*Pi],
  [ Newt(Sol1+<R*cos(t),R*sin(t)>)[1],
    Newt(Sol1+<R*cos(t),R*sin(t)>)[2],
    t = 0 .. 2*Pi]], colour=[red,blue], title = (radius=R));
```



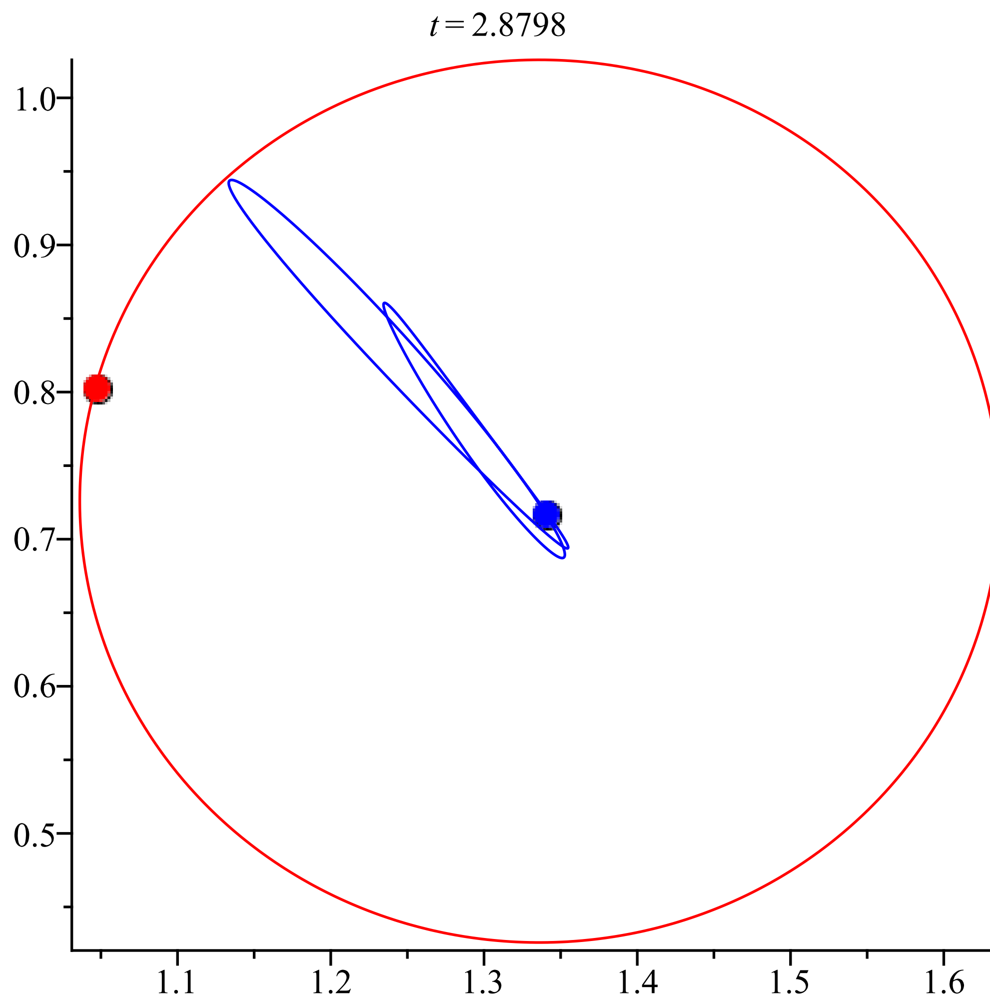
This time the blue curve extends outside the red circle, so we might not have convergence to Sol1 from points on this circle.
We could also try an animation.

```
> R:= 'R':
  animate(plot,[[[Sol1[1] + R*cos(t), Sol1[2] + R*sin(t),
    t = 0 .. 2*Pi],
   [ Newt(Sol1+<R*cos(t),R*sin(t)>)[1],
    Newt(Sol1+<R*cos(t),R*sin(t)>)[2],
    t = 0 .. 2*Pi]], colour=[red,blue]],R=0 .. 0.4);
```



Here's the red point moving around the circle of radius .3 centred at Sol1, while the blue point is is image under Newt.

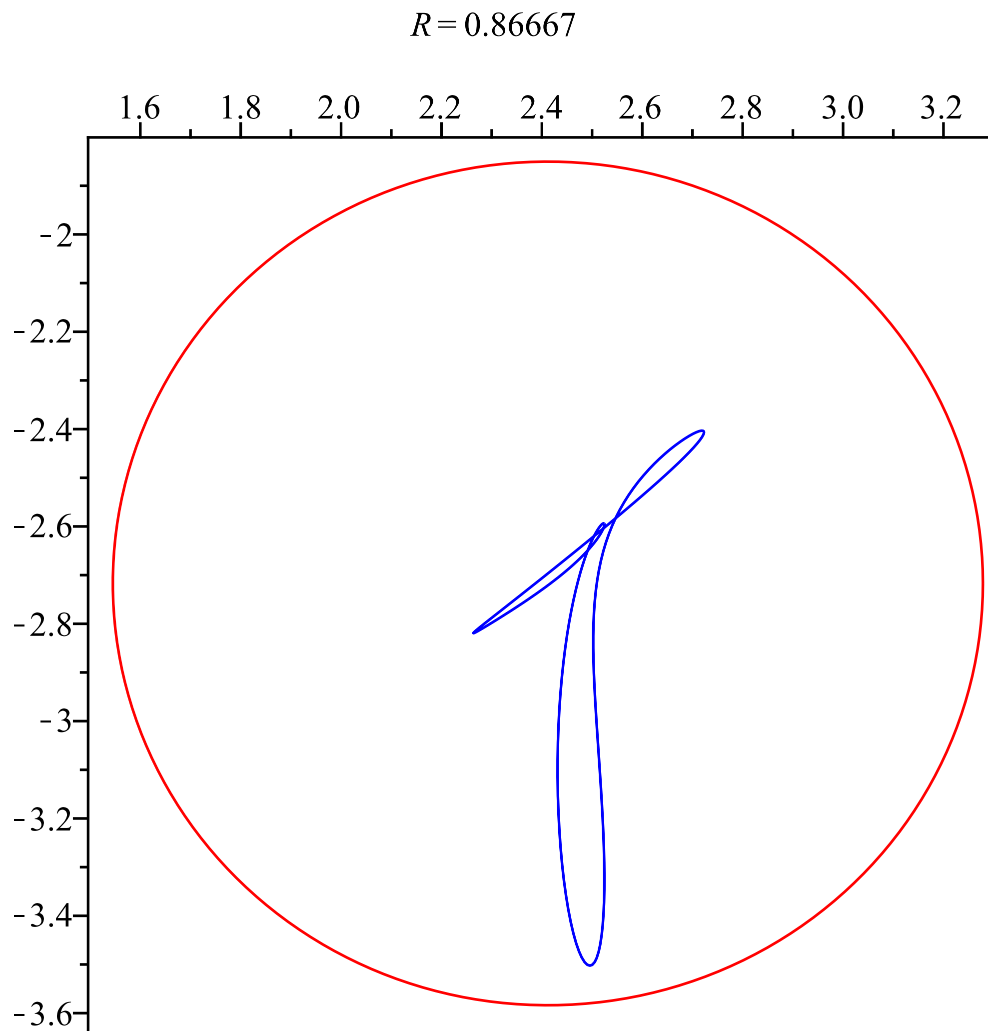
```
> animate(pointplot,[[[Sol1[1]+.3*cos(t), Sol1[2]+.3*sin(t)],
  [Newt(Sol1+<.3*cos(t),.3*sin(t)>)[1],
   Newt(Sol1+<.3*cos(t),.3*sin(t)>)[2]]],colour=[red,blue],
  symbol=solidcircle, symbolsize=10],t=0..2*Pi,background=P3);
```



It seems that the image of the circle is always inside the circle for $R < 0.3$, so Newton's method should converge to our first solution whenever the initial guess is within distance 0.3 of that point. Of course, this is only a lower bound, and it might converge to this point from points substantially farther away.

Similarly for Sol2:

```
> animate(plot,[[[Sol2[1] + R*cos(t), Sol2[2] + R*sin(t),
     t = 0 .. 2*Pi],
   [ Newt(Sol2+<R*cos(t),R*sin(t)>)[1],
     Newt(Sol2+<R*cos(t),R*sin(t)>)[2],
     t = 0 .. 2*Pi]], colour=[red,blue]],R=0.1 .. 0.9);
```



It seems that everything in the circle of radius 0.866 of Sol2 should converge to Sol2.

Approximating the basin of attraction

Here's another idea. Once we get within a distance 0.3 of our solution Sol1, we know we'll end up converging to it. Here's a function that takes x,y and n and returns the distance from $Newt^{(n)}(\langle x, y\rangle)$ to our solution.

```
> G:= proc(x,y,n)
  local P, count;
  Start at <x,y>
  P := <x,y>;
  Iterate Newt n times
  for count from 1 to n do P := Newt(P) end do;
  return the distance from P to Sol1
  sqrt((P[1]-Sol1[1])^2 + (P[2]-Sol1[2])^2)
  end proc;
```

$G := \textbf{proc}(x, y, n)$ **(2.1)**

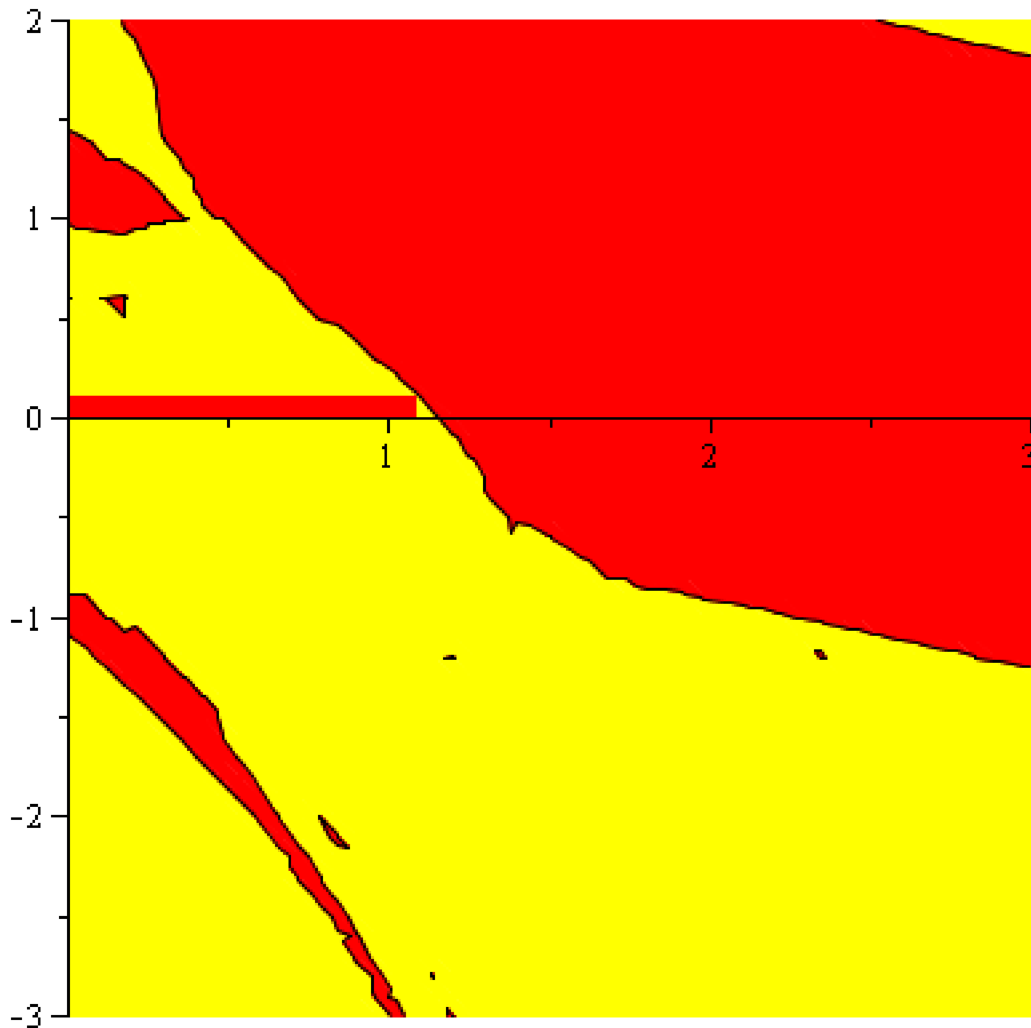
```
    local P, count;
    P := `<,>`(x, y);
    for count to n do P := Newt(P) end do;
    sqrt((P[1] - Sol1[1])^2 + (P[2] - Sol1[2])^2)
end proc
```

Now we could try an implicit plot of G(x,y,n) = 0.3 for some n, to get an approximation of part of the basin of attraction of our solution. But doing it this way would be a bad idea (recall the # "comments out" what's to the right of it, so it won't actually be executed):

```
> # implicitplot(G(x,y,3)=0.3, x = 0 .. 3, y = -3 .. 2);
```

The problem is that G(x,y,3) will be a very complicated symbolic expression. When I tried this Maple crashed. We just want to evaluate G with numbers for x and y, not symbolic x and y. This can be done by plotting the function (x,y) -> G(x,y,3) - 0.3 rather than the equation of expressions G (x,y,3) = 0.3.
When you do it this way, you must leave out the x = and y = from the intervals.

```
> implicitplot((x,y) -> G(x,y,3) - 0.3, 0 .. 3, -3 .. 2,
  gridrefine=1, filled=true);
```



The red region (and maybe some of the yellow) should be in the basin of attraction of our solution.
We could also try this for larger *n*, or even an animation of this for different values of *n*.
One way to make the procedure a little more efficient for larger *n* would be to allow for an early exit

from the loop: you might not need all n iterations to decide whether the point should or shouldn't be in the basin. If after k iterations you're within distance 0.3 of Sol1 you're definitely in the basin, if you're within distance 0.866 of Sol2 you're out.

```
> G:= proc(x,y,n)
  local P, count, d1, d2;
```

Start at <x,y>

```
  P := <x,y>;
```

Iterate up to n times

```
  for count from 1 to n do
```

Get the new point P

```
    P := Newt(P);
```

d1 is the square of the distance to Sol1

```
    d1:= (P[1]-Sol1[1])^2 + (P[2]-Sol1[2])^2;
```

If close enough to Sol1, you're in the basin; return that distance

```
    if d1 < 0.3^2 then return sqrt(d1) end if;
```

d2 is the square of the distance to Sol2

```
    d2:= (P[1]-Sol2[1])^2 + (P[2]-Sol2[2])^2;
```

If close enough to Sol2, you're outside the basin; return distance to Sol1

```
    if d2 < 0.866^2 then return sqrt(d1) end if;
  end do;
```

If you complete the loop without a decision, return distance to Sol1

```
  sqrt(d1);
  end proc;
```

$G :=$ **proc**(x, y, n) **(2.2)**

local P, $count$, $d1$, $d2$;

$P :=$ \`<,>\`(x, y);

for $count$ **to** n **do**

$P := Newt(P)$;

$d1 := (P[1] - Sol1[1])$^$2 + (P[2] - Sol1[2])$^2;

if $d1 < 0.3$^2 **then return** $\mathrm{sqrt}(d1)$ **end if**;

$d2 := (P[1] - Sol2[1])$^$2 + (P[2] - Sol2[2])$^2;

if $d2 < 0.866$^2 **then return** $\mathrm{sqrt}(d1)$ **end if**

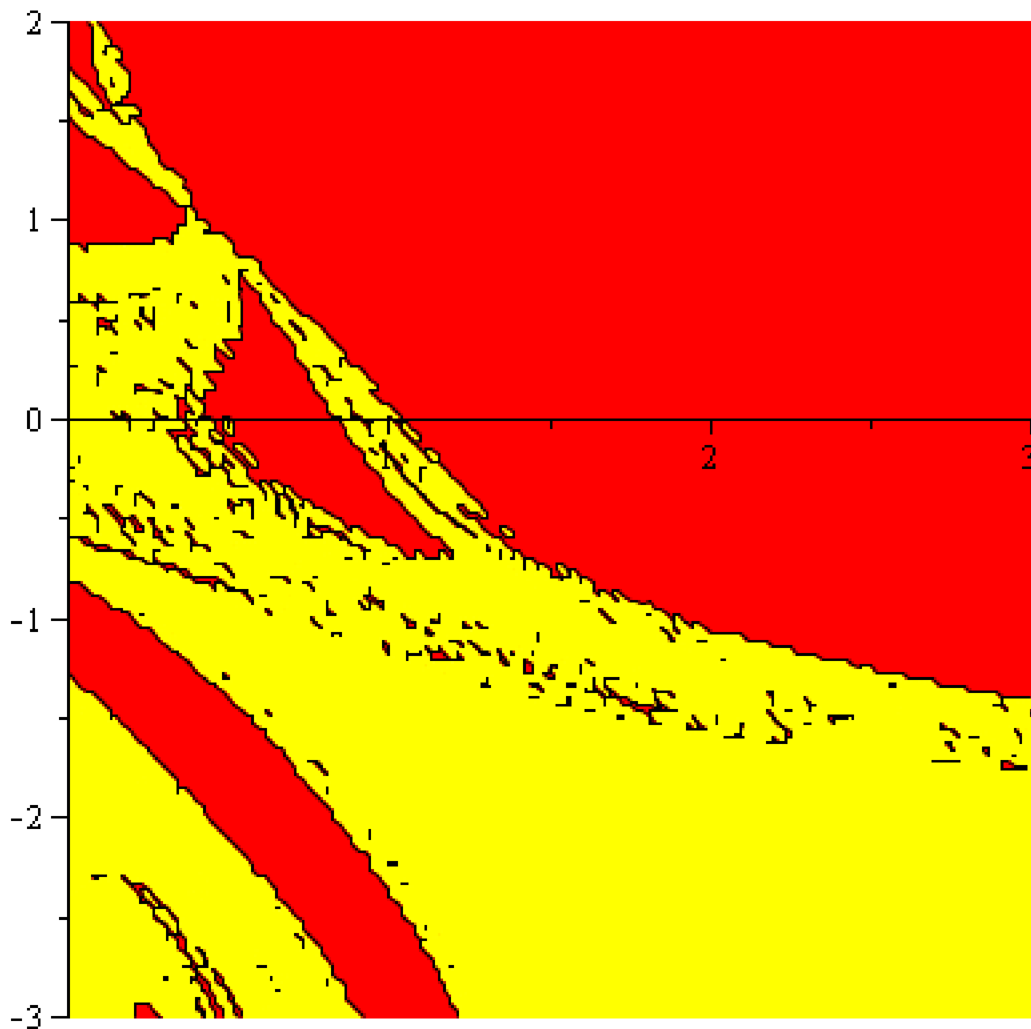
end do;

$\mathrm{sqrt}(d1)$

end proc

The next command takes a few minutes:

```
> implicitplot((x,y) -> G(x,y,10) - 0.3, 0 .. 3, -3 .. 2, grid=
  [40,40], gridrefine=2, filled=true);
```



Speeding it up

Here's perhaps a better way to see the basins of attraction of the two solutions.

First, I want a more efficient way to compute the Newton function.

Maple is an interpreted language. Programs written in Maple are stored, more or less in the form they are written in. To run the program, Maple must look at each line of code and call the appropriate procedures to do whatever it is supposed to do. Most of these procedures are themselves written in Maple, so each line of their code must also be interpreted. C, on the other hand, is a compiled language. After you write a C program, you give it to a C compiler which translates it into code that the computer's CPU can execute directly.

Being interpreted makes Maple very flexible, but compiled languages generally produce much faster programs: much of the time Maple spends in doing a calculation is taken up by interpreting the Maple code. And each time a line of code is executed (e.g. within a loop) it must be interpreted. A compiler just has to look at each line once. If it's in a loop, the same machine code will be run for each iteration.

Another factor in Maple's flexibility, but which also contributes to Maple's slowness, is that it is not a **strongly typed** language. There are lots of different types of data in Maple, e.g. integers, floating-point numbers, variable names, and all sorts of mathematical expressions, and Maple procedures have to be prepared to deal with all these types. So a lot of Maple's time is spent in checking on what type of data is being dealt with, in order to decide what to do with it. In a language such as C, things are usually much simpler: variables are declared as belonging to a certain type, so the compiler knows how to handle them.

The result of all this is that a Maple program might easily be 1000 times slower than a compiled C program that does the same thing. That's unfortunate, especially if you have to do a lot of calculations, e.g. for our basins of attraction.

Fortunately, there are ways of speeding up Maple in "number-crunching" applications: hardware floating point (**evalhf**) and **Compile**. In order to use these, a procedure can't do anything fancy such as matrix inversion or symbolic computation, just the standard mathematical operations and functions operating on numbers (and arrays of numbers). I'm going to use **Compile**. What this will do, given a suitable procedure, is produce a compiled version of that procedure that Maple can then call.

The **optimize** procedure in the **codegen** package tries to find the most efficient way of computing an expression (or, in our case, a Vector of expressions). For example, if it finds a subexpression that occurs in several places, it will only compute it once.

```
> Newt(<x,y>);
```

$$\left[\left[x-\frac{1.\cos(y)\,(x^2 y+x y^2-2.)}{2.\cos(y)\,x y+\cos(y)\,y^2-1.\,x^2-2.\,x y}+\frac{x\,(x+2.\,y)\,(x+\sin(y)-2.)}{2.\cos(y)\,x y+\cos(y)\,y^2-1.\,x^2-2.\,x y}\right],\left[y+\frac{x^2 y+x y^2-2.}{2.\cos(y)\,x y+\cos(y)\,y^2-1.\,x^2-2.\,x y}-\frac{1.\,y\,(2.\,x+y)\,(x+\sin(y)-2.)}{2.\cos(y)\,x y+\cos(y)\,y^2-1.\,x^2-2.\,x y}\right]\right] \quad \textbf{(2.1.1)}$$

```
> codegen[optimize](Newt(<X[1],X[2]>));
```

$$t1=\cos(X_2),\ t5=X_2^2,\ t7=X_1^2,\ t12=\frac{1}{2.\,t1\,X_1 X_2+t1\,t5-1.\,t7-2.\,X_1 X_2},\ t16=t7\,X_2+X_1\,t5-2.,\ t22=\sin(X_2),\ t24=t12\,(X_1+t22-2.),\ unknown_1=X_1-1.\,t1\,t12\,t16+X_1\,(X_1+2.\,X_2)\,t24,\ unknown_2=X_2+t12\,t16-1.\,X_2\,(2.\,X_1+X_2)\,t24 \quad \textbf{(2.1.2)}$$

I made this into a procedure that Compile could work with. Note that it puts its results in the same vector that it is given as input. This will also increase efficiency, because Maple won't need to use additional memory for each new result.

```
> N1:= proc(X)
  local X1, t1, t5, t7, t12, t16, t22, t24;
  X1:= X[1];
  t1 := cos(X[2]);
  t5 := X[2]^2;
```

```
t7 := X[1]^2;
t12 := 1/(2.*t1*X[1]*X[2]+t1*t5-t7-2.*X[1]*X[2]);
t16 := t7*X[2]+X[1]*t5-2.;
t22 := sin(X[2]);
t24 := t12*(X[1]+t22-2.);
X[1]:= X[1]-t1*t12*t16+X[1]*(X[1]+2.*X[2])*t24;
X[2]:= X[2]+t12*t16-X[2]*(2.*X1+X[2])*t24;
end proc;
```

$N1 := \mathbf{proc}(X)$ **(2.1.3)**

```
    local X1, t1, t5, t7, t12, t16, t22, t24;
    X1 := X[1];
    t1 := cos(X[2]);
    t5 := X[2]^2;
    t7 := X[1]^2;
    t12 := 1 / (2. * t1 * X[1] * X[2] + t1 * t5 − t7 − 2. * X[1] * X[2]);
    t16 := t7 * X[2] + X[1] * t5 − 2.;
    t22 := sin(X[2]);
    t24 := t12 * (X[1] + t22 − 2.);
    X[1] := X[1] − t1 * t12 * t16 + X[1] * (X[1] + 2. * X[2]) * t24;
    X[2] := X[2] + t12 * t16 − X[2] * (2. * X1 + X[2]) * t24
end proc
```

Now we give this procedure to Compile, which is in a package called Compiler.

```
> NC:= Compiler[Compile](N1);
```

$NC := \mathbf{proc}(\ )$ **(2.1.4)**

```
    option call_external, define_external(_m0c941d081c25653ec83d0b5a865edebe,
    MAPLE, LIB
     = "C:\Users\daddy\AppData\Local\Temp\daddy-4204\_m0c941d081c25653ec83d0b5\
    a865edebeU1yYq2xV.dll");
    call_external(0, 206377936, true, false, args)
end proc
```

NC must operate on an Array declared to have **datatype=float[8]**. These are a type of floating-point numbers that the computer's hardware operates on directly.

```
> X:= Array([1,2],datatype=float[8]);
```

$$X := \begin{bmatrix} 1. & 2. \end{bmatrix}$$ **(2.1.5)**

Like any Maple procedure, NC returns the last expression it computes, in this case the second coordinate of the new point.

```
> NC(X);
```

$$1.43264239736507926$$ **(2.1.6)**

But, as I said, it puts both coordinates in the Array that was its input.

```
> X;
```

$$\begin{bmatrix} 0.854598501646825 & 1.43264239736508 \end{bmatrix}$$ **(2.1.7)**

```
> Newt(<1,2>);
```

$$\begin{bmatrix} 0.8545985016 \\ 1.432642398 \end{bmatrix} \tag{2.1.8}$$

Actually I won't use this directly, but I'll write a new G that will iterate this up to 40 times, and will return -1 if it finds that the point $\langle x, y\rangle$ is attracted to the first solution, 1 if it's attracted to the second solution, 0 if we still don't know after 40 iterations.

```
> Sol11:= Sol1[1]; Sol12:= Sol1[2];
  Sol21:= Sol2[1]; Sol22:= Sol2[2];
```

$$Sol11 := 1.33623379100000$$
$$Sol12 := 0.725843022900000$$
$$Sol21 := 2.41188234700000$$
$$Sol22 := -2.71707384800000 \tag{2.1.9}$$

```
> G1:= proc(x,y)
  local count, d1, d2, X1, X2, Xt, t1, t5, t7, t12, t16, t22,
  t24;
  X1:= x; X2:= y;
  for count from 1 to 40 do
    t1 := cos(X2);
    t5 := X2^2;
    t7 := X1^2;
    t12 := 1/(2.*t1*X1*X2+t1*t5-t7-2.*X1*X2);
    t16 := t7*X2+X1*t5-2.;
    t22 := sin(X2);
    t24 := t12*(X1+t22-2.);
    Xt:= X1;
    X1:= X1-t1*t12*t16+X1*(X1+2.*X2)*t24;
    X2:= X2+t12*t16-X2*(2.*Xt+X2)*t24;
    d1:= (X1-Sol11)^2 + (X2-Sol12)^2;
    if d1 < 0.3^2 then return -1 end if;
    d2:= (X1-Sol21)^2 + (X2-Sol22)^2;
    if d2 < 0.866^2 then return 1 end if;
  end do;
  0;
  end proc;
```

G1 := **proc**(x, y) (2.1.10)

```
    local count, d1, d2, X1, X2, Xt, t1, t5, t7, t12, t16, t22, t24;
    X1 := x;
    X2 := y;
    for count to 40 do
        t1 := cos(X2);
        t5 := X2^2;
```

```
        t7 := X1^2;
        t12 := 1 / (2. * t1 * X1 * X2 + t1 * t5 − t7 − 2. * X1 * X2);
        t16 := t7 * X2 + X1 * t5 − 2.;
        t22 := sin(X2);
        t24 := t12 * (X1 + t22 − 2.);
        Xt := X1;
        X1 := X1 − t1 * t12 * t16 + X1 * (X1 + 2. * X2) * t24;
        X2 := X2 + t12 * t16 − X2 * (2. * Xt + X2) * t24;
        d1 := (X1 − Sol11)^2 + (X2 − Sol12)^2;
        if d1 < 0.3^2 then return −1 end if;
        d2 := (X1 − Sol21)^2 + (X2 − Sol22)^2;
        if d2 < 0.866^2 then return 1 end if
    end do;
    0
end proc
```

```
> GP:= Compiler[Compile](G1);
```

GP := proc( )
    option call_external, define_external(_m82aac686b8edef6a44a68608fff7e128,
    MAPLE, LIB
    = "C:\Users\daddy\AppData\Local\Temp\daddy-4204_m82aac686b8edef6a44a686\
    08fff7e128irbx50dp.dll");
    call_external(0, 206508656, true, false, args)
end proc **(2.1.11)**

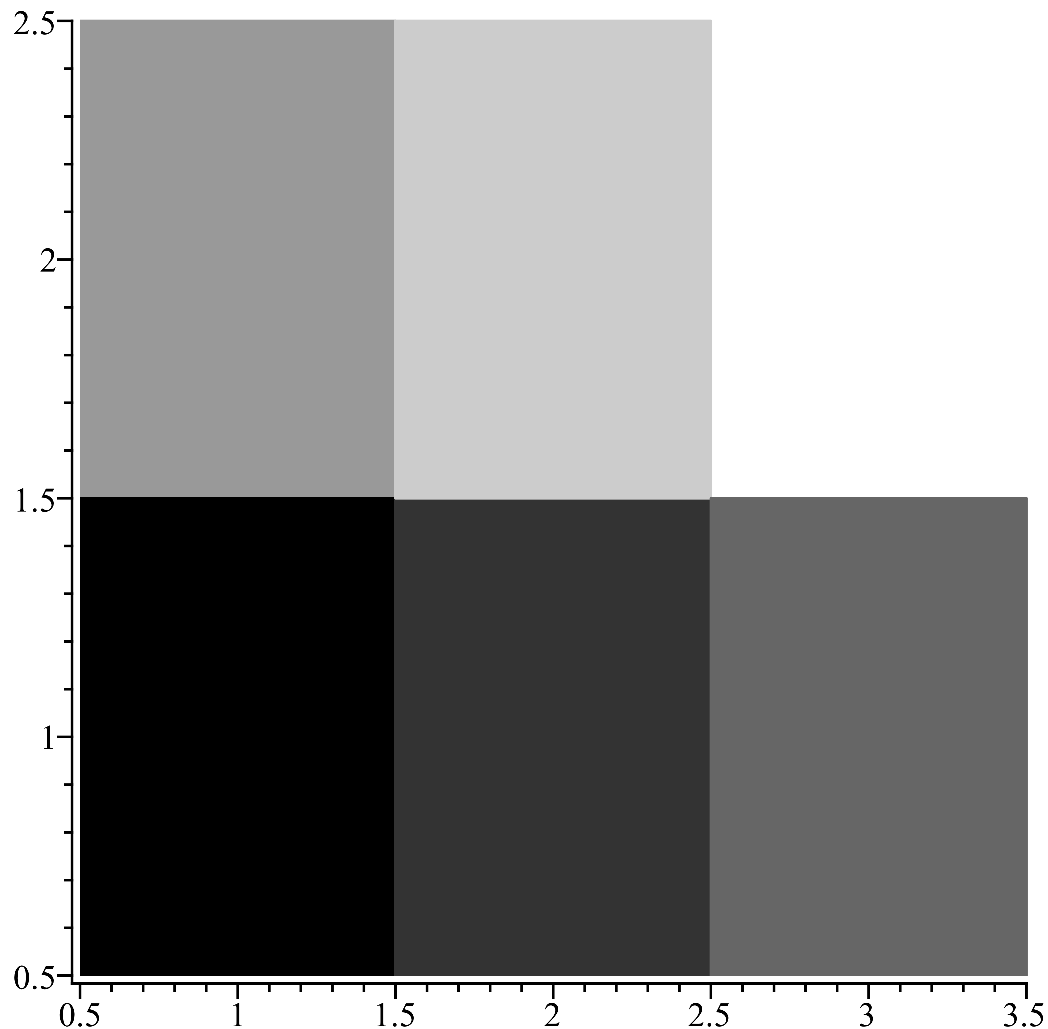
Now I'll compute G on a 500 by 500 grid, putting the results in a Matrix. The **datatype=float[8]** means that instead of the usual Matrix whose entries can be anything, this Matrix is only allowed entries that are 8-byte floating-point numbers. That makes handling it more efficient, especially with **Compile** and the plotting commands.

```
> M:= Matrix(500,500,(i,j) -> GP(i*3/500,-3+5*j/500),
  datatype=float[8]);
```

$$M := \begin{bmatrix} \textit{500 x 500 Matrix} \\ \textit{Data Type: float}_8 \\ \textit{Storage: rectangular} \\ \textit{Order: Fortran_order} \end{bmatrix} \quad \textbf{(2.1.12)}$$

To show the result, I'll use the **listdensityplot** function in the **plots** package. This takes our Matrix and produces a 2-dimensional representation of the entries of the Matrix: each entry corresponds to a rectangular cell, whose colour varies from black for the lowest value (in this case -1) to white for the highest (1). The *x* coordinate labels the row, the *y* coordinate the column.

```
> listdensityplot(<<1,2,3>|<4,5,6>>, style=patchnogrid);
```



The option **style=patchnogrid** removes the black lines between cells (which would really get in the way here because our cells will be very small). I also want to label the axes using the actual *x* and *y* values rather than rows and columns: the **tickmarks** option should let me do that.

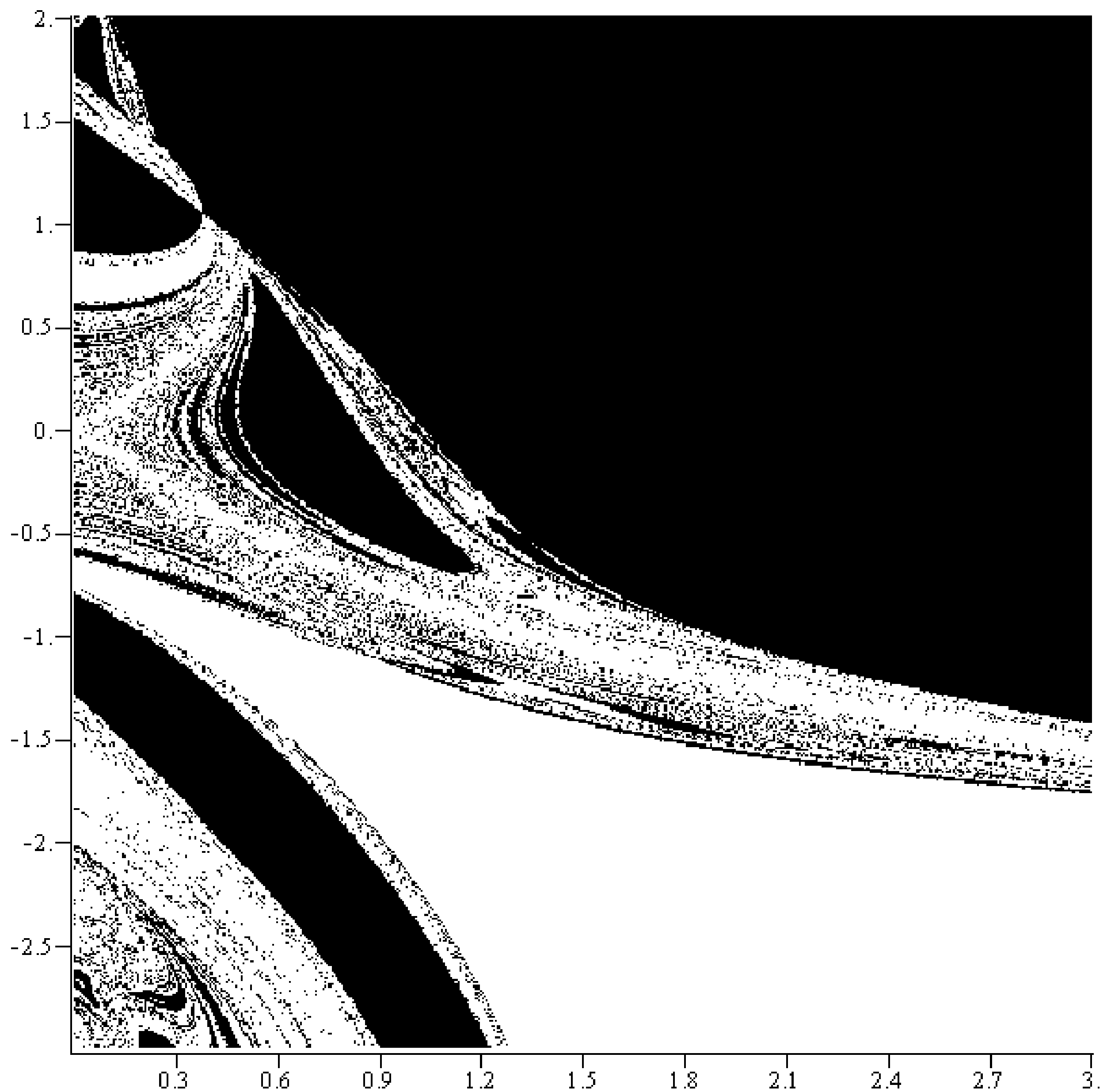
Now what I'd like to do is this:

```
> # listdensityplot(M, style=patchnogrid,
  # tickmarks =  [[seq(50*i = 0.3*i, i=0..10)],
  #  [seq(50*i = -3 + 0.5*i,i = 0 .. 10)]]);
```

It would work, except that Maple's graphical user interface is not happy with such a huge plot structure. Instead of plotting in the usual way, I'll send the output to a file. This also will use up more than the 100 MB limit. The path **d:/courses/m210/2010** should be replaced with whatever is appropriate for your computer. Note that / is used instead of \, even on a Windows machine.

```
> # plotsetup(gif, plotoutput="d:/courses/m210/2010/basins.
  gif",
  # plotoptions="height=600,width=600");
  # listdensityplot(M, style=patchnogrid,
  #   tickmarks =  [[seq(50*i = 0.3*i, i=0..10)],
  #    [seq(50*i = -3 + 0.5*i,i = 0 .. 10)]]);
  # plotsetup(default);
```

Here's the file imported into Maple.



Maple objects introduced in this lesson

#
filled = true (option for **implicitplot**)
return
optimize (in **codegen** package)
Compile (in **Compiler** package)
datatype = float[8] (option for **Matrix**)
listdensityplot (in **plots** package)
style = patchnogrid (option for **listdensityplot**)
tickmarks (option for plotting commands)

plotsetup
gif (graphics file format for **plotsetup**)