Drawing fractals in a few lines of Matlab

Thibaud Taillefumier

Disclaimer: This note is intended as a guide to generate fractal—and perhaps cool-looking—images using a few functionalities offered by Matlab. The broached topics are discussed as an excuse to get a hand on Matlab and are of no merits beyond that for the purpose of this class. This note is inspired from a class taught by Prof. Marcelo Magnasco at Rockefeller University.

In computational methods, one often wants to find the root of a differentiable function f, i.e. solve $f(x) = 0$. Suppose that one has a formula defining the function. For instance, f is the polynomial $f(x) = x^8 - 3x^3 + x^2 - 1$. Figure 1 shows that this polynomial has two real roots ≈ 0.7888 and ≈ -1.3309. For high-degree polynomials such as f, there is no formula giving the roots as functions of the polynomial's coefficients and one has to resort to numerical approximation. A very popular numerical method to find an approximated root is the Newton-Raphson algorithm. We are going to implement this method in Matlab and utilize this algorithm to generate fractal images in a few line of codes.

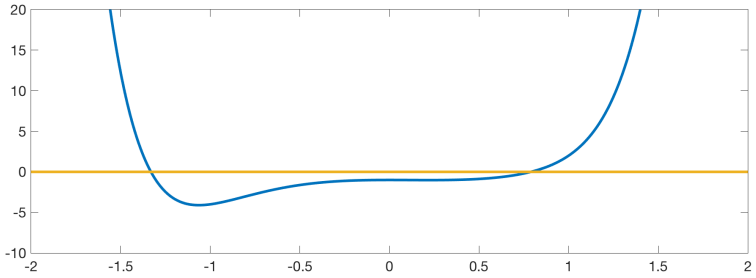

Figure 1: The graph of the polynomial $f(x) = x^8 - 3x^3 + x^2 - 1$ shows that there are two real roots ≈ 0.7888 and ≈ -1.3309.

The Newton-Raphson algorithm proceeds by gradual approximations of the root. Suppose, we start with a guess value x_0. Almost certainly, we have $f(x_0) \neq 0$ and x_0 is a poor approximation of a root. We want to find a rule to gradually update this guess to form a new guess x_1 that is hopefully closer to being a root than x_0.

The Newton-Raphson algorithm only utilizes $f(x_0)$ and $f'(x_0)$, the values taken by the function and its derivative at the current guess, and proceeds as follows: the new guess is chosen according to:

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}\,. \tag{1}$$

This is the value of the intercept of the tangent to the curve of f at x_0 with the x-axis. It can be shown that iterating guesses following this rule generally produces a sequence of numbers that converges toward a true root of f. Note that the Newton-Raphson algorithm supposes that one can compute the derivative, which is easy for polynomials: e.g. $f'(x) = 8x^7 - 9x + 2x$.

Let us implement the Newton-Raphson algorithm in Matlab. The first step is to implement a function that takes a guess and returns the next one:

```
function x1=NR_iter1(x0)

% compute the value taken by f at x0
f0=x0^8-3x0^3+x0^2-1;

% compute the value taken by f' at x0
df0=x0^7-9x0^2+2x0;

% compute the updated guess x1
x1=x0-f0/df0;
```

We can check that starting with a guess $x_0 = 2$ running the command

```
NR_iter1(x0)
```

returns 1.7396, which is not a good approximations of a root since $f(1.7396) \approx 95.6279$. However, we can check that iterating the command via nested execution calls, i.e.

```
NR_iter1(NR_iter1(x0))
NR_iter1(NR_iter1(NR_iter1(x0))) ...
```

gradually returns better approximation of a root. For instance, iterating 9 times yields 0.7888 for which $f(0.7888) \approx 0$. The Newton-Raphson algorithm works! The accuracy of the approximation is related to the number of guesses we are ready to make. The more guesses, the more accurate the estimate of the root. The following algorithm allows us to check that point using a loop structure rather than a nested execution call:

```
function x=NR_iter2(x0,nstep)

% set the current guess x to the initial guess x0
x=x0;

% loop iterating nstep guesses
for i=1:nstep

        % compute the value taken by f at x
        f=x^8-3x^3+x^2-1;

        % compute the value taken by f' at x
        df=x^7-9x^2+2x;

        % compute the updated guess x
        x=x-f/df;
end
```

To visualize the convergence of successive guesses, we can modify the above code to produce a figure. All is required is the introduction of a vector registering the guesses:

```
function x=NR_iter3(x0,nstep)

% initialize vector registering consecutive guesses
out=zeros(1,nstep+1);

% set the current guess x to the initial guess x0
x=x0
out(1)=x0;

% loop iterating nstep guesses
for i=1:nstep

        % compute the value taken by f at x
        f=x^8-3x^3+x^2-1;

        % compute the value taken by f' at x
        df=x^7-9x^2+2x;

        % compute the updated guess x
```

```
        x=x-f/df;

        % registering the guess
        out(1+i)=x;
end

% plot consecutive guesses
figure(1)
plot(out)
```

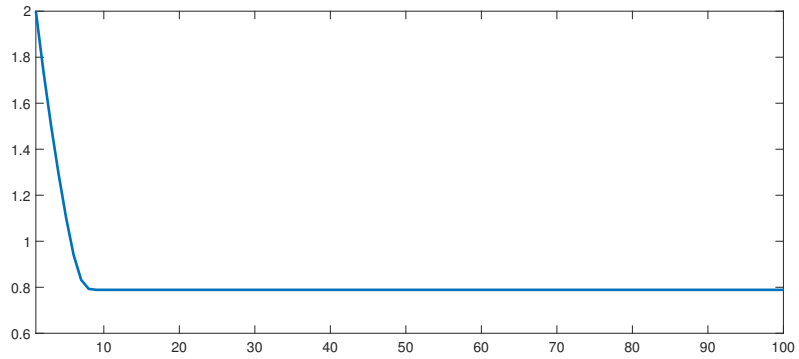

Figure 2: Convergence of 100 successive guesses starting from $x_0 = 2$ toward the root ≈ 0.7888.

Figure 2 confirms that successive guesses converge toward a true root ≈ 0.7888 when starting from $x_0 = 2$. However, we know that f has two roots and it is interesting to determine which roots the Newton-Raphson converges to depending on the initial condition x_0. Matlab allows us to investigate this point easily once we modify the code to take a vector of initial conditions as input: e.g. $\boldsymbol{x}_0 = -2, -1.99, -1.98, \ldots, 2$. We essentially need to adapt the code so that each arithmetic operation now acts on vectors component-wise. In Matlab this is done by using the **.** prefix in front of the arithmetic operations. For instance $(1, 2, 3).*(4, 5, 6) = (1 * 4, 2 * 5, 3 * 6)$: the components of the end vector are products of the components of the entry vectors:

```
function x=NR_iter4(x0,nstep)

% set the current guess x to the initial guess x0
```

```
x=x0;

% loop iterating nstep guesses
for i=1:nstep

        % compute the value taken by f at x
        f=x.^8-3x.^3+x.^2-1;

        % compute the value taken by f' at x
        df=x.^7-9x.^2+2x;

        % compute the updated guess x
        x=x-f./df;

        % registering the guess
        out(1+i)=x;
end

figure(1)
% plot consecutive guesses for different initial values
subplot(2,1,1)
plot(out)

% plot final guesses for different initial values
subplot(2,1,2)
plot(x0,x)
```

In the above code, we visualize the result of the algorithm using the subplot functionality that produces multi-panel graphs. The top panel of Figure 3 shows the convergence of the algorithm toward either root for different initial conditions. The bottom panel of Figure 3 shows the end guess after 100 iterations as a function of the initial condition. We can check that initial conditions close to a root value yield convergence toward that root. But for a range of initial conditions between the two roots, there are interleaved intervals for which the algorithm converges to distinct roots. The set of initial values for which the algorithm converges to a given root is called the basin of attraction of the root.

To generate the promised fractal pictures, we are going to extend the search for roots of the polynomial to the complex plane. In other words, we are going to consider that the argument of f is a complex number $z = x+iy$, where x is the real part of z, y is the imaginary part of z, and i is the imaginary unit ($i^2 = -1$). In the

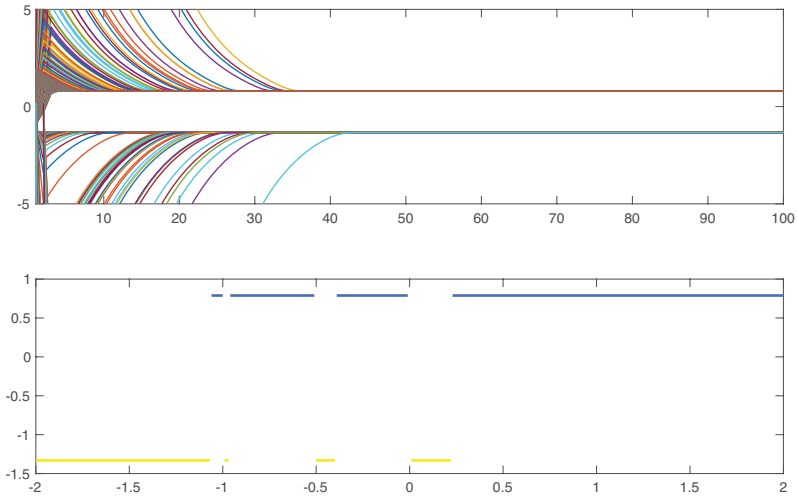

Figure 3: The top graph shows the convergence of the algorithm for different initial conditions $x_0 = -2, -1.99, -1.98, \ldots, 2$ toward the two roots ≈ 0.7888 (blue) and ≈ -1.3309 (yellow) after 100 iterations. The bottom graph shows the end guesses after 100 iterations depending as a function of the initial conditions. The identity of the root toward which the Newton-Raphson algorithm converges crucially depends on the initial condition.

complex plane, we expect polynomials to have as many roots as their degree with as many basins of attraction (i.e. 8 for f) [1]. Matlab can readily handle complex numbers such as $1 + 3i$, which are declared as follows:

```
z=1+1i*3
```

The evaluation and derivative operations are the same for polynomials when defined in the complex plane as when defined on the real line. As a consequence, we do not have to modify our existing Newton-Raphson algorithm to operate on complex numbers. We only need to generate a matrix of initial conditions representing a grid of complex numbers, e.g. $z_0 = x_0 + iy_0$ with $x_0 = -2, -1.99, -1.98, \ldots, 2$ and $y_0 = -2, -1.99, -1.98, \ldots, 2$. This can be done using the following function:

```
function x=ComplexGrid(xmin,xmax,ngrid)

% grid step size
delta=(xmax-xmin)/ngrid;

% range vector
x=xmin:delta:xmax;

% horizontal range matrix = real part
A=ones(nx+1,1)*x;

% vertical range matrix = imaginary part
B=x'*ones(1,nx+1);

% complex range matrix
M=A+1i*B;

figure(1)
% plot magnitude of complex numbers
subplot(1,2,1)
imagesc(abs(M))

% plot arguments of complex numbers
subplot(1,2,2)
imagesc(angle(M))
```

The above function produces images of the magnitude $|z|$ and argument $\arg z$ of the complex number z to make sure that we produce a grid (see Fig. 4). As expected,

[1] Note that f has only two roots on the real line.

the magnitude $|z|$ measures the distance from the origin of the grid, while the argument $\arg z$ measures the angle at the origin with respect to the real unit vector of the grid. Running the following Newton-Raphson algorithm for 100 iterations with a complex grid as an argument yields an outcome shown in Figure 5.

```
function x=NR_iter5(x0,nstep)

% set the current guess x to the initial guess x0
x=x0;

% loop iterating nstep guesses
for i=1:nstep

        % compute the value taken by f at x
        f=x.^8-3x.^3+x.^2-1;

        % compute the value taken by f' at x
        df=x.^7-9x.^2+2x;

        % compute the updated guess x
        x=x-f./df;

        % plot magnitude of complex numbers
        subplot(1,2,1)
        imagesc(abs(x))

        % title indicating frame guesses
        str=sprintf('frame %d',i);
        title(str);

        % plot arguments of complex guesses
        subplot(1,2,2)
        imagesc(angle(x))

        pause(0.1)
end

figure(1)
% plot magnitude of final complex guesses
subplot(1,2,1)
```

```
imagesc(abs(x))

% plot arguments of final complex guesses
subplot(1,2,2)
imagesc(angle(x))
```

Figure 5 depicts the different basins of attraction by labelling initial conditions z_0 with the magnitude and the argument of the complex roots obtained via Newton-Raphson iterations. One can check that we recover 8 different basins of attraction. The horizontal mid-line in Figure 5 (not shown) represents the initial conditions that are real numbers. One can check that we recover the results obtained by considering only real numbers as the midline runs over exactly two distinct basins of attraction (still represented in yellow and blue, albeit with different shades of blue in the left and right panels). Amazingly, the frontier between distinct basins of attraction defines a very complicated curves that exhibit a self-similar structure. One can check by eyes that the same motifs tends to appear at different scales if one zooms in on the frontier between distinct basins. These fractal images are examples of Julia sets.

Figure 4: Representation of the complex plane $z = x + iy = re^{i\theta}$ in Matlab. Left panel: magnitude $r = |z|$. Right panel: angle $\theta = \arg(z)$

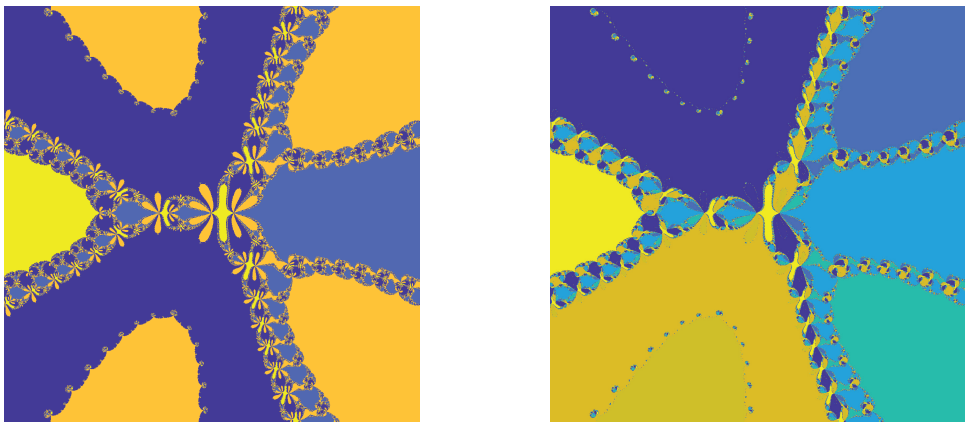

Figure 5: Basin of attractions: different initial conditions in the complex plane may converge toward different complex roots. Left panel: colormap of the magnitude of 100-th Newton-Raphson iteration for different initial conditions in the complex plane. Right panel: colormap of the angle of 100-th Newton-Raphson iteration for different initial conditions in the complex plane. The yellow and blue large components around the mid-horizontal axis are associated with the two real roots. The other components are associated with other complex roots.