Newton's method for complex polynomials

A preprint version of a “Mathematical graphics” column from *Mathematica in Education and Research*.

Mark McClure

mcmcclure@unca.edu

Department of Mathematics
University of North Carolina at Asheville
Asheville, NC 28804

Abstract

Newton's method is one of the most venerable and important algorithms in numerical analysis. It can also be used to produce some beautiful illustrations of chaos. It turns out that the structure of these images point the way to a recently published algorithm to find *all* the roots of a complex polynomial.

Initialization

1. Introduction

1.1 The basics

Newton's method is a technique to estimate roots of functions. Newton originally applied the technique to polynomials and these are the functions we will focus on as well. Thus, we wish to find the solutions to $p(z) = 0$, where p is a polynomial.

The technique starts with a guess z_1 and uses an iterative procedure to find a sequence of improved guesses. In particular, if z_n is “close” to a root, then

$$z_{n+1} = z_n - \frac{p(z_n)}{p'(z_n)}$$

will generally be (much) closer. Given an initial guess z_1, the above equation recursively defines a sequence which hopefully converges to a root of p. For example, we may use the following code to find an approximation to $\sqrt{2}$.

```
p[z_] := z^2 - 2;
n[z_] = z - p[z] / p'[z];
FixedPointList[n, 1.0]
{1., 1.5, 1.41667, 1.41422, 1.41421, 1.41421, 1.41421}
```

A derivation of Newton's method may be found in almost any calculus text and many details may be found in almost any numerical analysis text.

1.2 Global behavior

Can we use Newton's method to find *all* the roots of a complex polynomial? It seems that this very natural question was first asked by Arthur Cayley in 1879. He also indicated an important first step towards a solution when he asked: Given a complex polynomial p and a starting point for Newton's method, to which root does the sequence converge? Cayley went further and proved that the sequence converges to the closest root when p is a quadratic polynomial. He also indicated that the question seemed to be difficult when p is a cubic polynomial.

Difficult indeed! Such an algorithm has only been discovered in the last 20 years and a reasonably efficient algorithm has only been recently discovered. In fact, the main objective of this column is to indicate how computer graphics can illuminate a key idea in the algorithm published in [1]. In this beautiful and well-written paper, the authors apply a wide range of advanced mathematical tools to prove their algorithm works. The paper is highly recommended to anyone interested in applications of complex dynamics.

The essential difficulty is that Newton's method is inherently a *local* procedure. Given a particular polynomial together with some information (perhaps a graph) describing the approximate location of a root, it is usually fairly easy to apply Newton's method to find a highly accurate approximation of the root. However, the global structure of Newton's method can be quite complicated and an understanding of this global structure is essential to the algorithm. In an effort to understand this global structure, we turn to computer graphics.

2. Images of Newton's method

2.1 Generating a basic picture

The basic strategy to generate a picture illustrating the global behavior of Newton's method for a polynomial p is as follows. We choose a rectangular region of the complex plane, frequently a region containing all the roots of p. We finely subdivide the region into rectangles, say 300×300 of them, each corresponding to a complex number. We then perform Newton's method from each of these complex numbers and color the corresponding square according to which root the sequence converges. If the sequence does not converge to a root, a real possibility, then we color the rectangle black.

We will illustrate the technique on the simplest possible cubic, $p(z) = z^3 - 1$. In spite of the simplicity of this particular example, we would like to develop code which is applicable to a broad range of examples. This adaptability is the main reason that the following code is more complicated than necessary for this one example.

Here is the polynomial together with it's roots.

```
p[z_] := z^3 - 1;
theRoots = z /. NSolve[p[z] == 0, z]
```

```
{-0.5 - 0.866025 i, -0.5 + 0.866025 i, 1.}
```

We will be iterating the function $n(z) = z - p(z)/p'(z)$ many times, so we write a compiled version of it. We will also be checking $|p(z)|$ with each iteration as a stopping criterion ($|p(z)|$ small indicates that we are close to a root), so it will be good to have a compiled version of p as well.

```
cp = Compile[{{z, _Complex}},
   Evaluate[p[z]]];
n = Compile[{{z, _Complex}},
   Evaluate[z - p[z] / p'[z]]];
```

We now generate a table of orbits obtained by iterating **n** until **p** is small in absolute value, or a maximum of 100 iterations is reached.

```
bail = 100;
orbitData = Table[
   NestWhileList[n, x + ⅈ y,
    Abs[cp[#]] > 0.0001 &,
    1, bail],
   {y, -1.5, 1.5, 0.01},
   {x, -1.5, 1.5, 0.01}];
```

In order to color the initial points, we need to know to which root each orbit converges. While that is very easy for this particular example, it can be a much more delicate question in general. Clearly, we would say that a term z_n in a sequence generated by Newton's method is close to a root z_0 if $|z_n - z_0|$ is small. But how small is necessary? Lemma 15 of [1] suggests that $|3\,p(z)/p'(z)|$ is a good choice, at least when z_0 has multiplicity one. This is the basis of the **sameRootFunc** function below. Unfortunately, there appears to be no such criterion easily applicable to roots of higher multiplicity. *Note:* A root z_0 is said to have multiplicity m if $(z - z_0)^m$ is a factor of $p(z)$, but $(z - z_0)^{m+1}$ is not a factor.

We would also like to shade the initial points according to how fast the orbit converges. Thus, the **whichRoot** command below returns the length of the orbit as well as the index of the root. Finally, note that the **Union** in the definition of **numRoots** allows the code to deal with multiple roots.

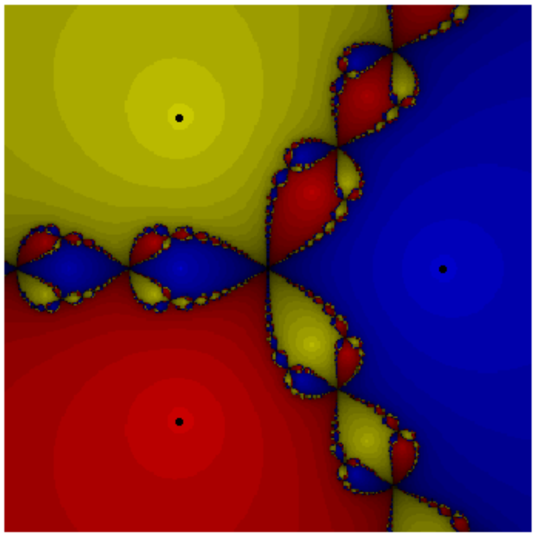
```
numRoots = Length[Union[theRoots]];
sameRootFunc = Compile[{{z, _Complex}},
   Evaluate[Abs[3 p[z] / p'[z]]]];
whichRoot[orbit_] := Module[
   {i, z},
   z = Last[orbit];
   i = 1;
   Scan[If[Abs[z - #] < sameRootFunc[z],
      Return[i], i++] &, theRoots];
   If[i ≤ numRoots,
    {i, Length[orbit]},
    None]
  ];
rootData = Map[whichRoot, orbitData, {2}];
```

Next, we would like to turn **rootData** into a matrix of color directives for display by **RasterArray**. This is rather easily done for the general polynomial using **Hue** and this will be the default approach taken by our program. But we would also like a technique to input a list of colors. The following code uses **RGBColor** with the placeholder **cc** to indicate colors. For example, the color yellow is indicated by **RGBColor[cc,cc,0]**.

```
colorList = {RGBColor[cc, 0, 0],
   RGBColor[cc, cc, 0], RGBColor[0, 0, cc]};
cols = rootData /. {
    {k_Integer, l_Integer} :>
     (colorList[[k]] /. cc -> (1 - l / (bail + 1))^8),
    None → RGBColor[0, 0, 0]};
```

Finally, we display the image together with the roots. Note that the locations of the roots need to be scaled to the integer coordinates used by **RasterArray**.

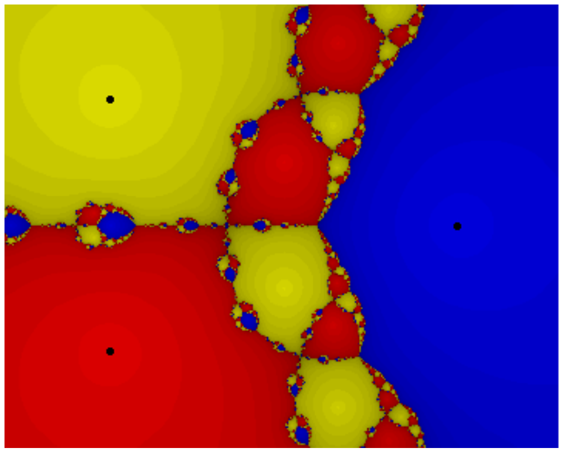
```
rootPoints = theRoots /. z_?NumericQ :>
    Point[{100 Re[z] + 150, 100 Im[z] + 150}];
Show[Graphics[{
    RasterArray[cols],
    PointSize[0.015], rootPoints}],
  AspectRatio → Automatic];
```



2.2 More examples

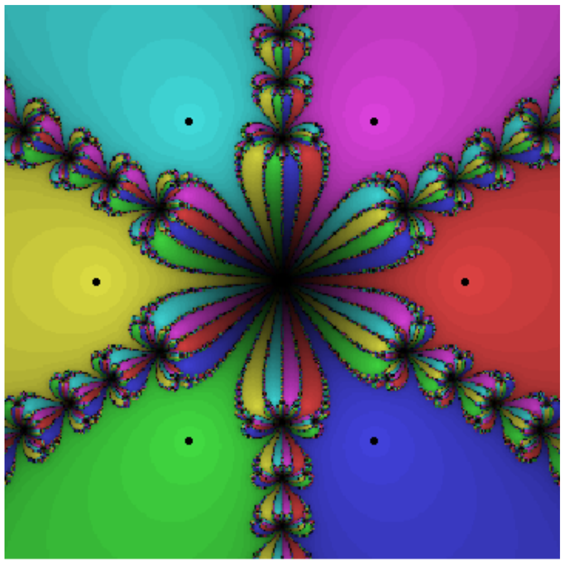
We would like to apply this idea to a wide range of examples. Thus the above algorithm has been encoded in the command **NewtonPic** defined in the initialization cells. For example, here's the picture corresponding to the polynomial $p(z) = z^3 - 2\ z - 5$, the very polynomial Newton used to illustrate his method. Note that the bounds of the region plotted are described by two complex numbers representing the lower left and upper right corners of the region in question. Also, the command accepts an option **Colors** which should be a list of colors indicated using **RGBColor** directives as we did before.

```
NewtonPic[z^3 - 2 z - 5, {z, -2 - 2 ⅈ, 3 + 2 ⅈ},
  Colors → {RGBColor[cc, 0, 0],
    RGBColor[cc, cc, 0], RGBColor[0, 0, cc]}];
```



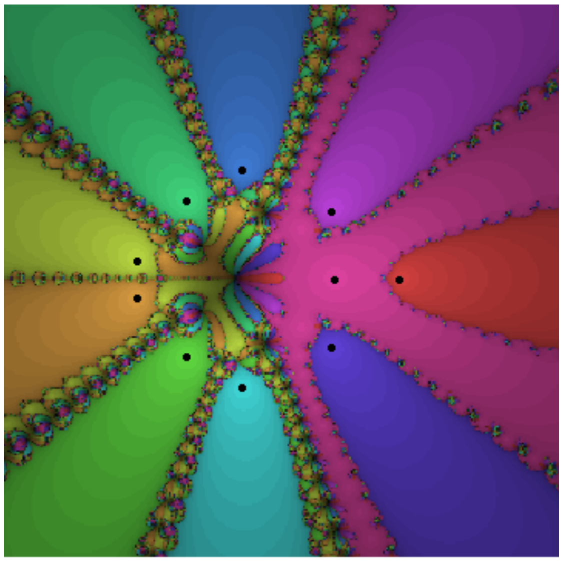
If we don't specify **Colors**, the command uses **Hue** to automatically generate colors. This is nice when more colors are being used, as in the following example.

```
NewtonPic[z^6 - 1, {z, -1.5 - 1.5 ⅈ, 1.5 + 1.5 ⅈ}];
```



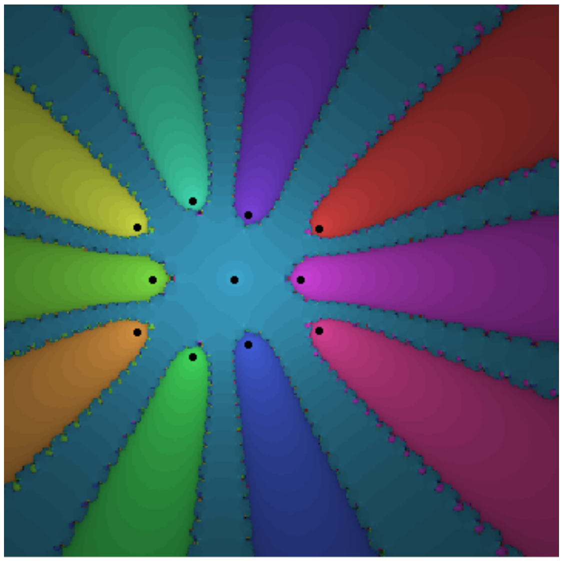
Next, we apply the technique to a randomly generated polynomial.

```
SeedRandom[8];
p[z_] = 4 + Sum[Random[Integer, {-2, 2}] z^k, {k, 0, 10}];
NewtonPic[p[z], {z, -2.5 - 3 I, 3.5 + 3 I}];
```



Finally, we apply the technique to a random polynomial with a root of multiplicity two at the origin. Note that roots of higher multiplicities tend to cause the program problems.

```
SeedRandom[1];
p[z_] = z^2 Sum[Random[Integer, {-2, 2}] z^k, {k, 0, 10}];
NewtonPic[p[z], {z, -2.5 - 3 I, 3.5 + 3 I}];
```



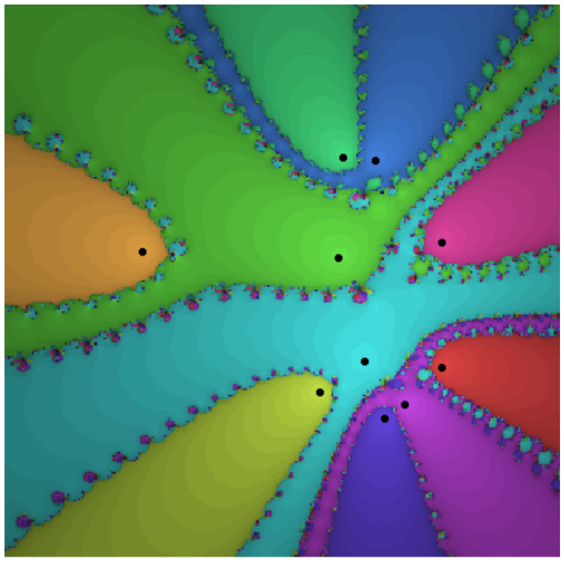
2.3 More focused examples

Our objective is to understand the basics of the algorithm to find all the roots of a complex polynomial described in [1]. The key idea is based on the fact that all these pictures have certain structural similarities. In particular, associated with each root is at least one *channel to infinity*; i.e. it is always possible to draw a curve starting at a point and continuing to ∞ without ever leaving the basin of attraction for that point. Let's illustrate this with a few more examples. First, we'll see what happens if we pack a large number of roots in the unit circle. We can generate such a polynomial by writing a function **RandomPointInCircle**, which generates a point uniformly inside the unit circle, and then forming a product of terms of the form ($z - z_0$), where each z_0 is generated using **RandomPointInCircle**.

```
RandomPointInCircle := Module[{a, b},
   a = b = 1;
   While[a^2 + b^2 > 1,
    a = Random[Real, {-1, 1}];
    b = Random[Real, {-1, 1}]];
   a + b I];
```

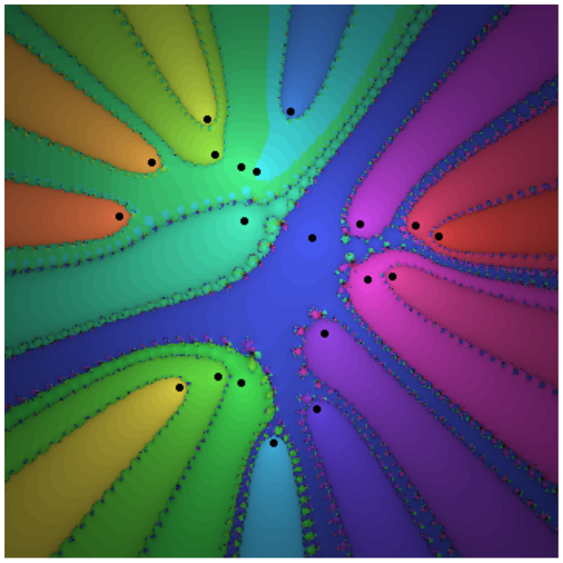
Here's the corresponding picture with 10 roots.

```
SeedRandom[1];
p[z_] = Product[z - RandomPointInCircle, {10}];
NewtonPic[p[z], {z, -1.5 - 1.5 I, 1.5 + 1.5 I}];
```



Here's the picture with 20 roots. Note that the limits of the program are being tested here and it helps to increase the resolution.

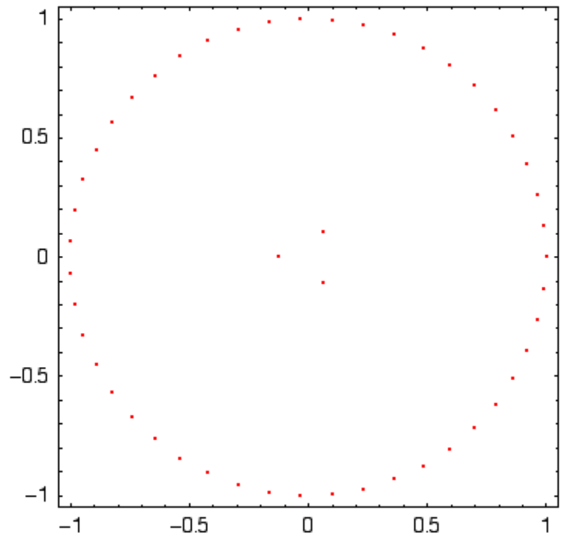
```
SeedRandom[3];
p[z_] = Product[z - RandomPointInCircle, {20}];
NewtonPic[p[z], {z, -1.5 - 1.5 I, 1.5 + 1.5 I},
  Resolution → 400];
```



Now suppose we'd like to generate a polynomial with a small number of roots deep in the interior of the unit circle surrounded by a large number of roots on the unit circle. Here is such a polynomial with 50 roots. It seems that if any polynomial could violate the “channel to infinity” observation, then this one would. Note that we could express the polynomial more simply by replacing the product with the expression $z^{47} - 1$, but the pictures aren't quite as nice using that formula.

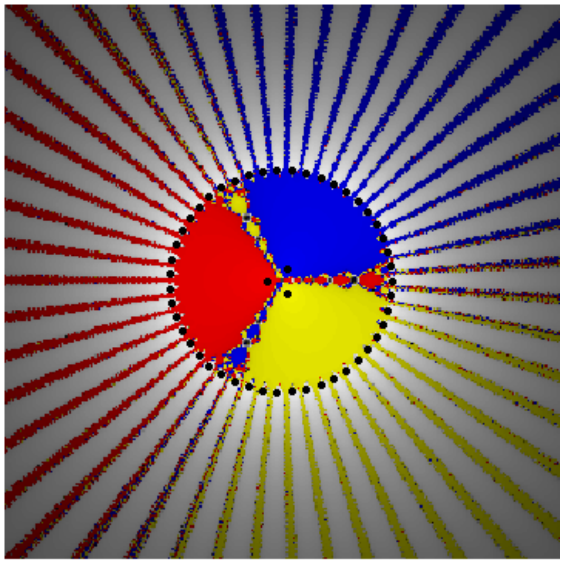
Here is the polynomial and its roots.

```
p[z_] := ((8 z)^3 + 1) Product[z - e^(2 π i θ), {θ, 1 / 47, 1, 1 / 47}];
theRoots = z /. NSolve[p[z] == 0, z];
ListPlot[{Re[#], Im[#]} & /@ theRoots,
  AspectRatio → Automatic,
  PlotJoined → False];
```



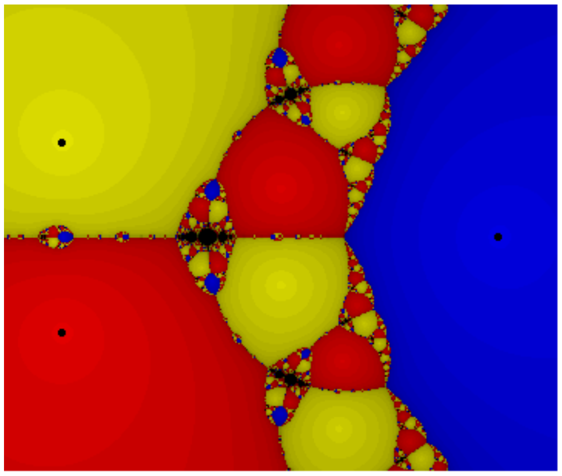
We now generate the corresponding picture. Note that we shade all the roots on the unit circle a similar shade of gray and the ones in the interior with the three primary colors. In order to do so, we need the appropriate **colorList** for the **Colors** option. The **colorList** should be specified in the same order as the roots are returned by **NSolve** after duplicates are removed. A simple way to accomplish this is to write a function which returns the desired color in terms of the root. This function can then be mapped on to the roots, as we've done here. Reducing the default **Tolerance** and increasing **MaxIterations** improves the quality of this picture.

```
chooseColor[z_] := Which[
   Abs[z] > 1 / 2, RGBColor[cc, cc, cc],
   Abs[z] < 1 / 2 && Re[z] < 0, RGBColor[cc, 0, 0],
   Abs[z] < 1 / 2 && Re[z] > 0 && Im[z] < 0, RGBColor[cc, cc, 0],
   Abs[z] < 1 / 2 && Re[z] > 0 && Im[z] > 0, RGBColor[0, 0, cc]];
colorList = chooseColor /@ theRoots;
surroundPic = NewtonPic[p[z], {z, -2.5 - 2.5 ⅈ, 2.5 + 2.5 ⅈ},
   Colors → colorList,
   Tolerance → 0.01, MaxIterations → 500];
```



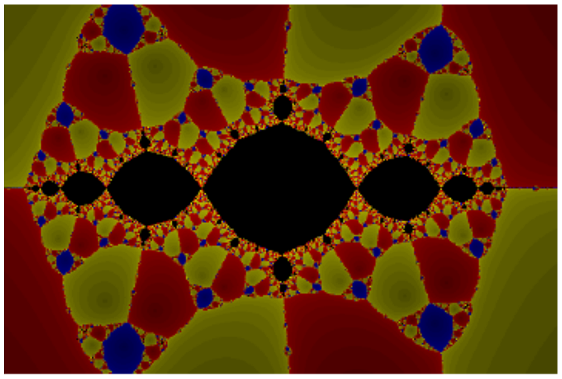
Finally, we look at a beautiful, but somewhat disappointing example which illustrates a potential difficulty with Newton's method.

```
NewtonPic[z^3 - 0.64310316 z - 0.35689684,
  {z, -0.7 - 0.8 I, 1.2 + 0.8 I},
  Colors → {RGBColor[cc, 0, 0],
    RGBColor[cc, cc, 0],
    RGBColor[0, 0, cc]}];
```



This image appears to contain a solid black structure. We can zoom in to get a closer look. Note that we change each **cc** to **1-cc**. This flips the intensities of the colors, increasing the contrast near the boundary of the black region.

```
NewtonPic[z^3 - 0.64310316 z - 0.35689684,
  {z, -0.12 - 0.08 I, 0.12 + 0.08 I},
  Colors → {RGBColor[1 - cc, 0, 0],
    RGBColor[1 - cc, 1 - cc, 0], RGBColor[0, 0, 1 - cc]},
  Resolution → 400];
```



Clearly very pretty, but disappointing. The black region consists of complex points whose orbits do not approach *any* root. As the picture indicates, it is quite possible for this region to have positive area. In fact, complex points chosen from the interior of the black region lead to orbits attracted to stable periodic orbits. These points need to be avoided when applying Newton's method to approximate a root.

3. Finding all the roots of a complex polynomial

3.1 The `NewtonSolve` command

There is a function called **`NewtonSolve`** defined in the initialization cells of this notebook. This command is based on the algorithm described in [1]. **`NewtonSolve`** accepts a polynomial as its first argument and a second argument to indicate the variable. It returns a list of root and multiplicity pairs as illustrated in the following command.

```
p[z_] := ((2 z)^3 - 1)^2 (z^7 - 1);
sols = NewtonSolve[p[z], z]
```

```
{{1. + 0. i, 1}, {0.5 + 7.31473 × 10^-149 i, 2}, {0.62349 + 0.781831 i, 1},
 {-0.25 + 0.433013 i, 2}, {-0.900969 - 0.433884 i, 1}, {-0.25 - 0.433013 i, 2},
 {-0.222521 - 0.974928 i, 1}, {0.62349 - 0.781831 i, 1},
 {-0.900969 + 0.433884 i, 1}, {-0.222521 + 0.974928 i, 1}}
```

Note that 10 distinct roots have been found, three of which have multiplicity 2. We can use the following command to check that the **`sols`** are indeed approximate roots.

```
Max[Abs[p /@ (First /@ sols)]]
```

```
3.67857 × 10^-14
```

Comment: It is certainly *not* the objective of this column to propose an alternative to **`NSolve`** for polynomials, which is implemented in the kernel and has been tested worldwide for nearly 20 years. The purpose of **`NewtonSolve`** is simply to implement an interesting, recently discovered algorithm for expository purposes. Given these considerations, no attempt has been made to test **`NewtonSolve`** extensively.

3.2 The algorithm

The **`NewtonSolve`** code is fairly complicated and deals with several subtleties. The reader interested in all the details may consult the code together with the main reference [1]. A basic outline is sketched in this section.

The essential idea is to construct a set of complex numbers S_d with the following property. Given any polynomial p of degree d and any root z_0 of p, there is some element z_1 of S_d which gives rise to sequence converging to z_0 if z_1 is chosen as a starting point for Newton's method applied to p. We emphasize that S_d should depend only on the degree d and not on the particular polynomial p.

While it is not possible to construct S_d at quite this level of generality, it is possible to construct S_d, if we assume that p is suitably normalized. In particular, we assume that all of the roots of p lie inside the unit disk $D = \{z : |z| < 1\}$. This is not a genuine restriction. Given a polynomial p, it is not difficult to find a number M such that all roots of p lie in a disk of radius M centered at the origin. Thus p may be rescaled so that all its roots lie in the unit disk.

Now, given a polynomial p of degree d the basic outline is as follows, *assuming* we can find the set S_d of initial points and the number M which bounds the roots of p.

- Set up the collection of initial points S_d.
- Normalize p to obtain the polynomial p_M, where $p_M(z) = p(M z)$.
  All the roots of p_M will then lie in the unit disk.
- Using the points in S_d as initial seeds for Newton's method, find all the roots of p_M.
- Multiply each of the roots found in the previous step by M.

Further details on each step are as follows.

• The set S_d of initial points

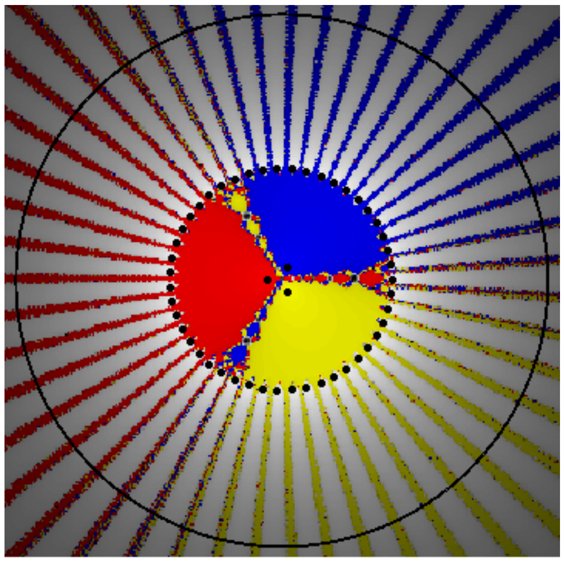
This is the key step illuminated by computer graphics. The graphics indicate that at least one channel to infinity is present for every root. The authors of [1] *prove* this fact. Even more, they prove that for each root of any normalized polynomial, there is at least one channel to infinity of a minimum width. This suggests the possibility that S_d may be constructed by uniformly distributing sufficiently many points on a circle of sufficient radius. While this approach can work, the authors are able to reduce the size of S_d by using multiple circles, since the width of the channels can vary. Ultimately, the authors arrive at the following definition of S_d.

```
ring[r_, d_] := With[{n = Ceiling[8.32547 d Log[d]]},
   Flatten[Transpose[Partition[Table[r Exp[2 Pi I j / n],
       {j, 0, n - 1}], Ceiling[n / d], Ceiling[n / d], 1, {{}}]]]];
S[d_] := With[{R = 1 + √2 , s = Ceiling[0.26632 Log[d]]},
   Flatten[Table[ring[R (1 - 1 / d) ^ ((2 ν - 1) / (4 s)), d],
     {ν, 1, s}]]];
```

Note that **ring[r,d]** constructs a ring of approximately 8.33 d log(d) points equally distributed about a circle of radius **r**. The points are ordered so that the arguments of consecutive points differ by about $2\,\pi/d$. While any ordering would do, the authors report that this seems most efficient. Then **s** constructs approximately 0.266 log(d) of these rings. Thus S_d contains approximately $2\ d\ \log(d)^2$ points.

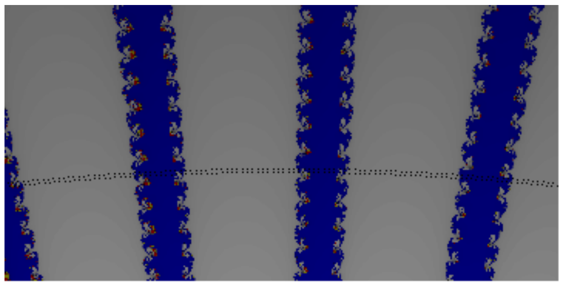
Here is how S_d fits inside one of our more interesting pictures.

```
thePoints = {Re[#], Im[#]} & /@ S[50];
Show[{surroundPic, Graphics[
    {PointSize[0.001],
     Point /@ (thePoints /. x_?NumericQ →
         301 (x + 2.5) / 5)}]},
  AspectRatio → Automatic];
```



Here is a closer look.

```
p[z_] := ((8 z)^3 + 1) Product[z - ⅇ^(2 π ⅈ θ), {θ, 1 / 47, 1, 1 / 47}];
NewtonPic[p[z], {z, -0.5 + 2.0 ⅈ, 0.5 + 2.5 ⅈ},
  Colors → colorList, Resolution → 200,
  Tolerance → 0.01, MaxIterations → 500,
  Epilog → {PointSize[0.001],
    Point /@ (thePoints /. {x_, y_} →
        {400 (x + 0.5), 200 (y - 2)})}];
```



• Normalizing *p*

In order to normalize *p*, we need an easily computable, apriori upper bound on the maximum size of a root of *p*. The **NewtonSolve** program uses Cauchy's estimate (5.3.3 of [2]) which states that if $p(z) = a_n z^n + \cdots + a_1 z + a_0$ and $p(z_0) = 0$, then

$$\left| z_0 \right| \le \max\left\{ \frac{|a_{n-1}|}{|a_n|}, \ldots \frac{|a_0|}{|a_n|} \right\} + 1.$$

Note that this estimate is very general and, therefore, not always sharp. This can be problematic for the program. Consider the following example.

```
K = 32;
p[z_] = Sum[2^k z^(K-k), {k, 0, K}];
```

$$p[z\_] = \sum_{k=0}^{K} 2^k z^{K-k};$$

Cauchy's estimate leads to the upper bound of 2^{32} for the size of a root, but there is another test specifically for polynomials with positive coefficients (5.3.6(c) of [2]) which leads to the rather sharp upper bound of 2 for the same polynomial. We can specify the upper bound using the **NormalizationFactor** option to improve performance. In this case, the upper bound of 2 rather than 2^{32} makes the difference between finding the solutions and crashing.

```
NewtonSolve[p[z], z,
 NormalizationFactor → 2]
```

```
{{1.96386 + 0.378502 ⅈ, 1}, {1.85674 + 0.743325 ⅈ, 1},
 {1.68251 + 1.08128 ⅈ, 1}, {1.44747 + 1.38016 ⅈ, 1}, {1.16011 + 1.62915 ⅈ, 1},
 {0.83083 + 1.81926 ⅈ, 1}, {0.471518 + 1.94362 ⅈ, 1}, {0.0951638 + 1.99773 ⅈ, 1},
 {-0.28463 + 1.97964 ⅈ, 1}, {-1.57211 + 1.23632 ⅈ, 1}, {-1.30972 + 1.5115 ⅈ, 1},
 {-1. + 1.73205 ⅈ, 1}, {-1.99094 - 0.190112 ⅈ, 1}, {-1.91899 - 0.563465 ⅈ, 1},
 {-1.77767 - 0.916453 ⅈ, 1}, {-1.57211 - 1.23632 ⅈ, 1}, {-1.30972 - 1.5115 ⅈ, 1},
 {-1. - 1.73205 ⅈ, 1}, {-0.654136 - 1.89 ⅈ, 1}, {-0.28463 - 1.97964 ⅈ, 1},
 {0.0951638 - 1.99773 ⅈ, 1}, {0.471518 - 1.94362 ⅈ, 1}, {0.83083 - 1.81926 ⅈ, 1},
 {1.16011 - 1.62915 ⅈ, 1}, {1.44747 - 1.38016 ⅈ, 1}, {1.68251 - 1.08128 ⅈ, 1},
 {1.85674 - 0.743325 ⅈ, 1}, {1.96386 - 0.378502 ⅈ, 1}, {-1.91899 + 0.563465 ⅈ, 1},
 {-0.654136 + 1.89 ⅈ, 1}, {-1.77767 + 0.916453 ⅈ, 1}, {-1.99094 + 0.190112 ⅈ, 1}}
```

• Find all the roots of p_M

To find all the roots of p_M, we specify a particular error tolerance **`tol`** and iterate Newton's method function until p_M is less than **`tol`** in absolute value. One subtlety involves the appropriate maximum number of iterations. The authors suggest

$$\texttt{maxIters} = \texttt{Ceiling}\left[\frac{\texttt{Log}\left[1+\sqrt{2}\right] - \texttt{Log}[\texttt{tol}]}{\texttt{Log}[\texttt{d}] - \texttt{Log}[\texttt{d}-1]}\right];$$

and this is the choice of **`maxIters`** used by the program.

• A final detail

The final step in the outline is straightforward. There are several other subtle details, however. One particularly interesting detail is the identification of the multiplicity of each root. This is an important practical concern, because usually all roots are found well before all the initial points in S_d are used. Of course, a polynomial of degree d has d roots, if counted according to multiplicity. Thus, when there are multiple roots, we need to know their multiplicities so we can stop when all roots have been found. It turns out that the rate of convergence indicates the multiplicity of the root. To see this, consider Newton's method as applied to the polynomial $p(z) = z^m$. In this case, $N(z) = z - z^m / \left(m\, z^{m-1}\right) = \frac{m-1}{m} z$. Thus the iterates of N converge to zero exponentially. More precisely, $N^k(z_1) = \left(\frac{m-1}{m}\right)^k z_1$. **`NewtonSolve`** uses this observation to classify the multiplicity of each root.

References

[1] Hubbard, J., Schleicher, D., and Sutherland, S. “How to find all roots of complex polynomials by Newton's method.” *Invent. Math.* **146** #1 (2001), 1-33.

[2] Barbeau, E. J. *Polynomials*. Springer-Verlag, NY 1989.