Mandelbrot set

From Techniques for computer generated pictures in complex dynamics

Jump to: navigation, search

Here we will give a few algorithms and show the corresponding results.

Contents

[hide]

The Mandelbrot set

Certainly, Wikipedia's page about this set in any language should be a good introduction. Here I give a very quick definition/reminder:

The Mandelbrot set, denoted M, is the set of *complex numbers* cc such that the *critical point* z=0z=0 of the *polynomial* $P(z)=z2+cP(z)=z2+c$ has an *orbit* that is not attracted to infinity. If you do not know any of the italicized words, go and look on the Internet.

It is significant for two reasons:

- The *Julia set* of PP is *connected* if and only if $c\in Mc\in M$.
- The dynamical system $z\mapsto P(z)z\mapsto P(z)$ is *stable* under a perturbation of PP if and only if $c\in\partial Mc\in\partial M$, where $\partial\partial$ is a notation for the *topological boundary*.

Bibliography

Methods presented here are either direct translation or enhancement of algorithms that I learned first in the 1980's from elementary programs found in popular journals about computers, in the mid 1990's in the book *The beauty of Fractals*, and then when I started a career as a mathematician in the late 1990's directly from discussion with colleagues, especially Xavier Buff, Christian Henriksen and John H. Hubbard.

A major proportion of the main ideas and algorithms are already present in *The science of Fractal Images*.

The Mu-Ency by Robert Munafo contains a more detailed discussion of some of the algorithms.

- Peitgen, Richter, *The beauty of Fractals*, 1986, Springer-Verlag, Heidelberg.
- Peitgen, Saupe (Editors), *The Science of Fractal Images*, 1987, Springer-Verlag.
- Mu-Ency - *The Encyclopedia of the Mandelbrot Set*, 1996-2016 Robert P. Munafo.
- wikibooks.org: Fractals

Drawing algorithms

All the algorithms I will present here are scanline methods.

Basic algorithm

The most basic is the following, it is based on the following theorem:

Theorem: *The orbit of 0 tends to infinity if and only if at some point it has modulus >2.*

This theorem is specific to $z \mapsto z^2+c$z↦z2+c, but can be adapted to other families of polynomials by changing the threshold 22 to another one. Here the threshold does not depend cc but in other families it may.

Now here is the algorithm:

```
Choose a maximal iteration number N
For each pixel p of the image:
  Let c be the complex number represented by p
  Let z be a complex variable
  Set z to 0
  Do the following N times:
    If |z|>2 then color the pixel white, end this loop prematurely, go to the
next pixel
    Otherwise replace z by z*z+c
  If the loop above reached its natural end: color the pixel p in black
  Go to the next pixel
```

I am not going to use the syntax above again, since it is too detailed. Let us see what it gives in a Python-like style:

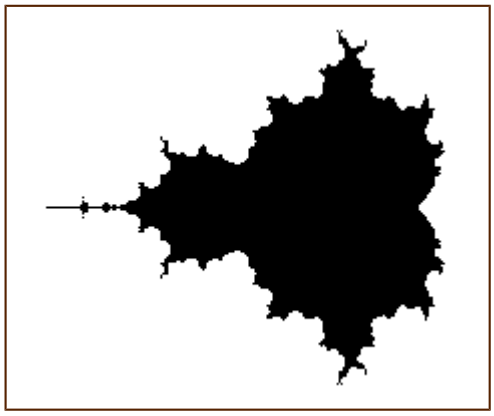
```
for p in allpixels: # replace here by your own loop or pair of loops to scan all
pixels
  c = p.affix # here you may replace by your code computing c (complex nb)
  z = 0j
  color = black # 'color' will be assigned to p at the end, black is a temporary
value
  for n in range(0,N):
    if squared_modulus(z)>4:
      color = white
      break # this will break the innermost for loop and jump just after (two
lines below)
    z = z*z+c
  p.color = color # so it will be black unless we ran into the line color=white
```

Note that if $z=x+iy$z=x+iy then its squared modulus is x^2+y^2x2+y2 whereas $|z|=\sqrt{x^2+y^2}$|z|=x2+y2. Not only working with squared_modulus saves one step: taking a square root, but also this step is usually a time consuming one.

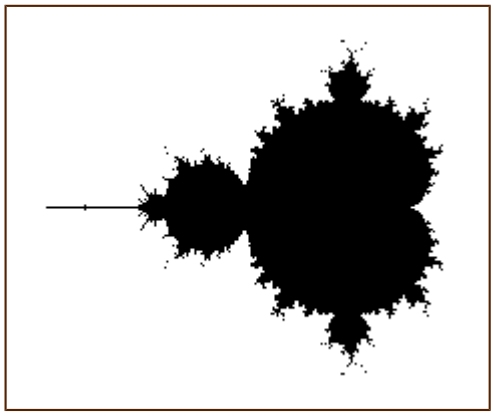
Here in a C++ like style: (std::complex<double> simplified into complex)

```
for(int i=0; i<height; i++) {
  for(int j=0; j<width; j++) {
    complex c = some formula of i and j;
    complex z = 0.;
    for(int n=0; n<N; n++) {
      if(squared_modulus(z)>4) {
        image[i][j]=black;
        goto label;
      }
      z = z*z+c;
    }
    image[i][j]=white;
    label: {}
  }
}
```

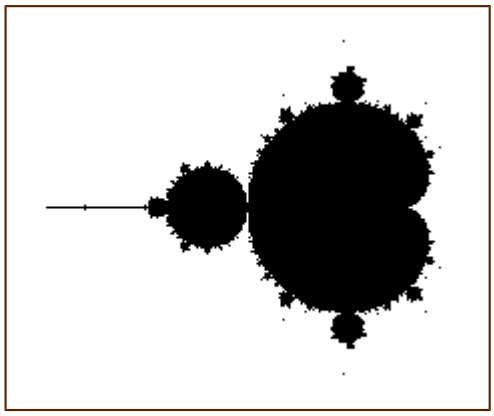
Let us now look at the kind of pictures we get with that. We took an image size of 241×201 pixels, with a mathematical width of 3.0 and so that the center pixel has affix -0.75. The only varying parameter between them is the maximal number of iterations N.

- 

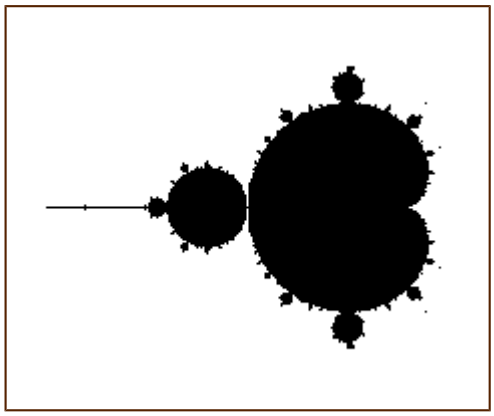
N=10

- 

N=20

- 

N=100

- 

N=1 000 000

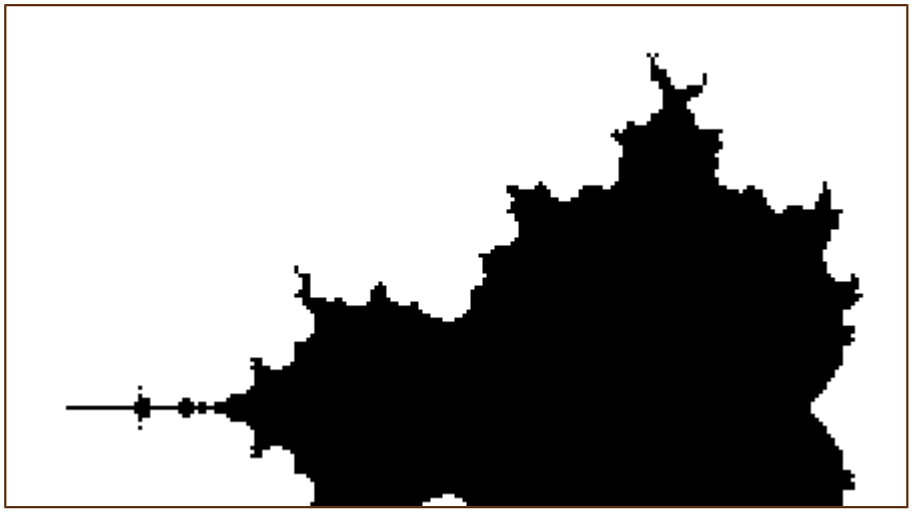
All images except the last one took a fraction of a second to compute on a modern laptop. Back then in the 1980's it was different.

Ideally the maximal number of iterations N should be infinity, but then the computation would never stop. The idea is then that the bigger N is, the more accurate the picture should be.

The N=one million image took 45s to compute on the same laptop, which is pretty long given today's computers power. What happens? Every black pixel requires 106106 iterations, and since there are 9 771 of them, we do at least ~10101010 times the computation z*z+c. To accelerate this, one should find a way to detect that we are in M so that we use less than the maximal number of iterations.

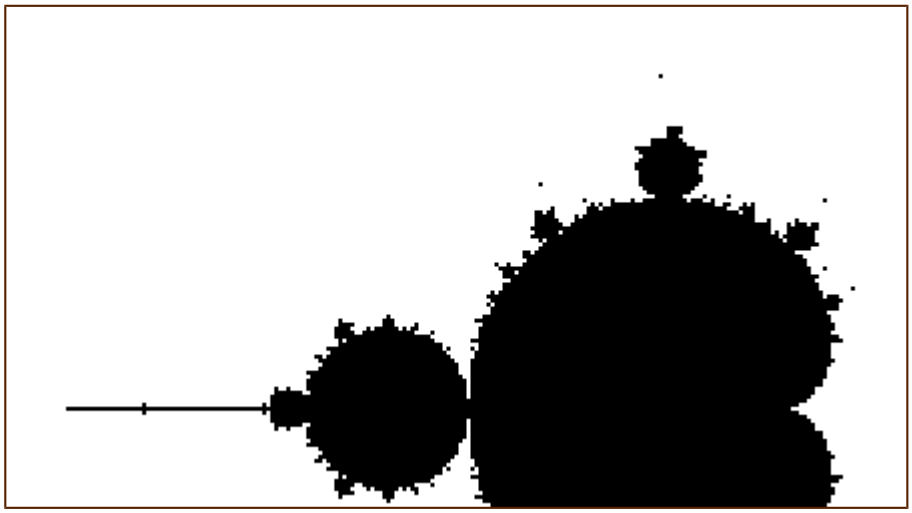
But that is not the only problem with increasing N. Not only it took much time, but the difference with N=100 is pretty small. Worse: it seems that by increasing N we are losing parts of the picture. What happens is that we are testing pixel centers or corners, and there are strands of M that are thin and wiggle between the pixel centers, so that as soon as N is big enough, the pixel gets colored white.

Here are enlarged versions of parts of these images, so that pixels are visible.

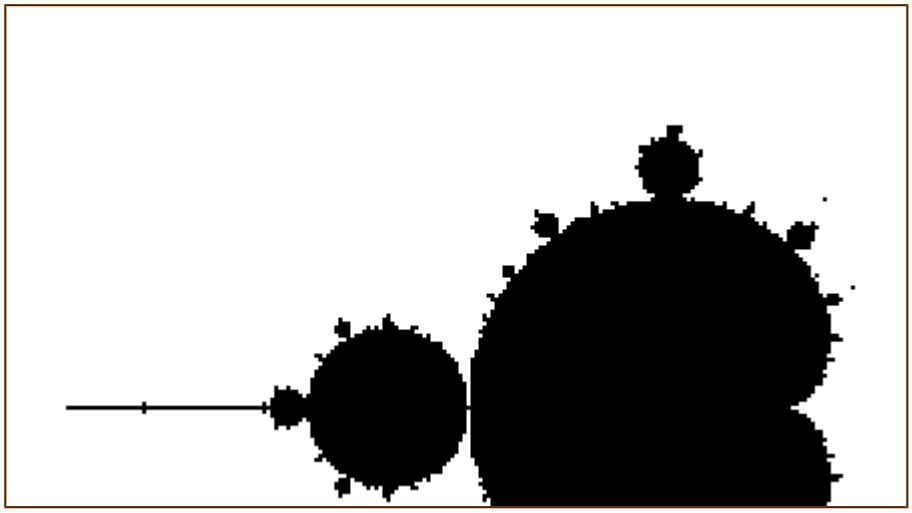


N=10

N=20

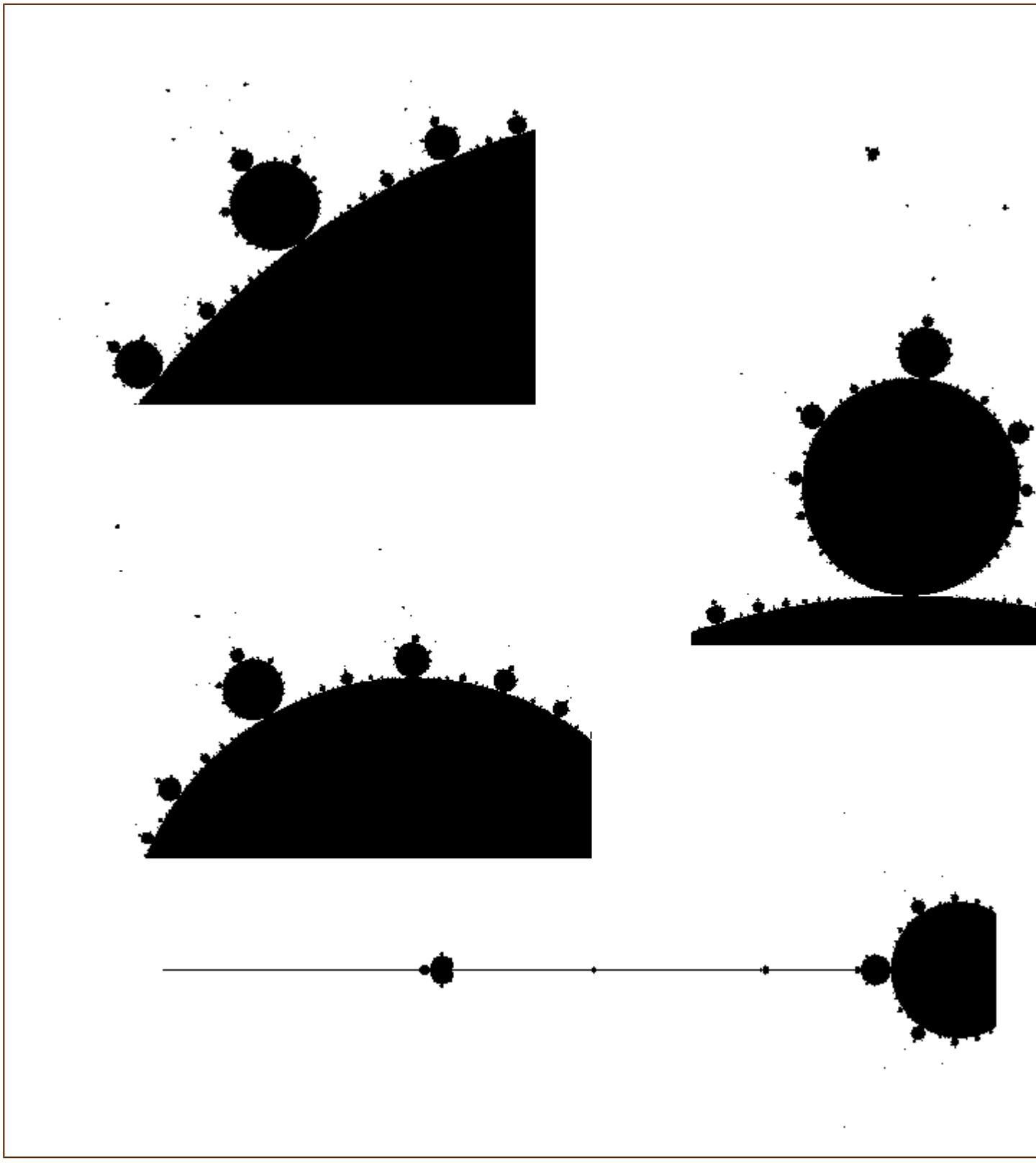


N=100

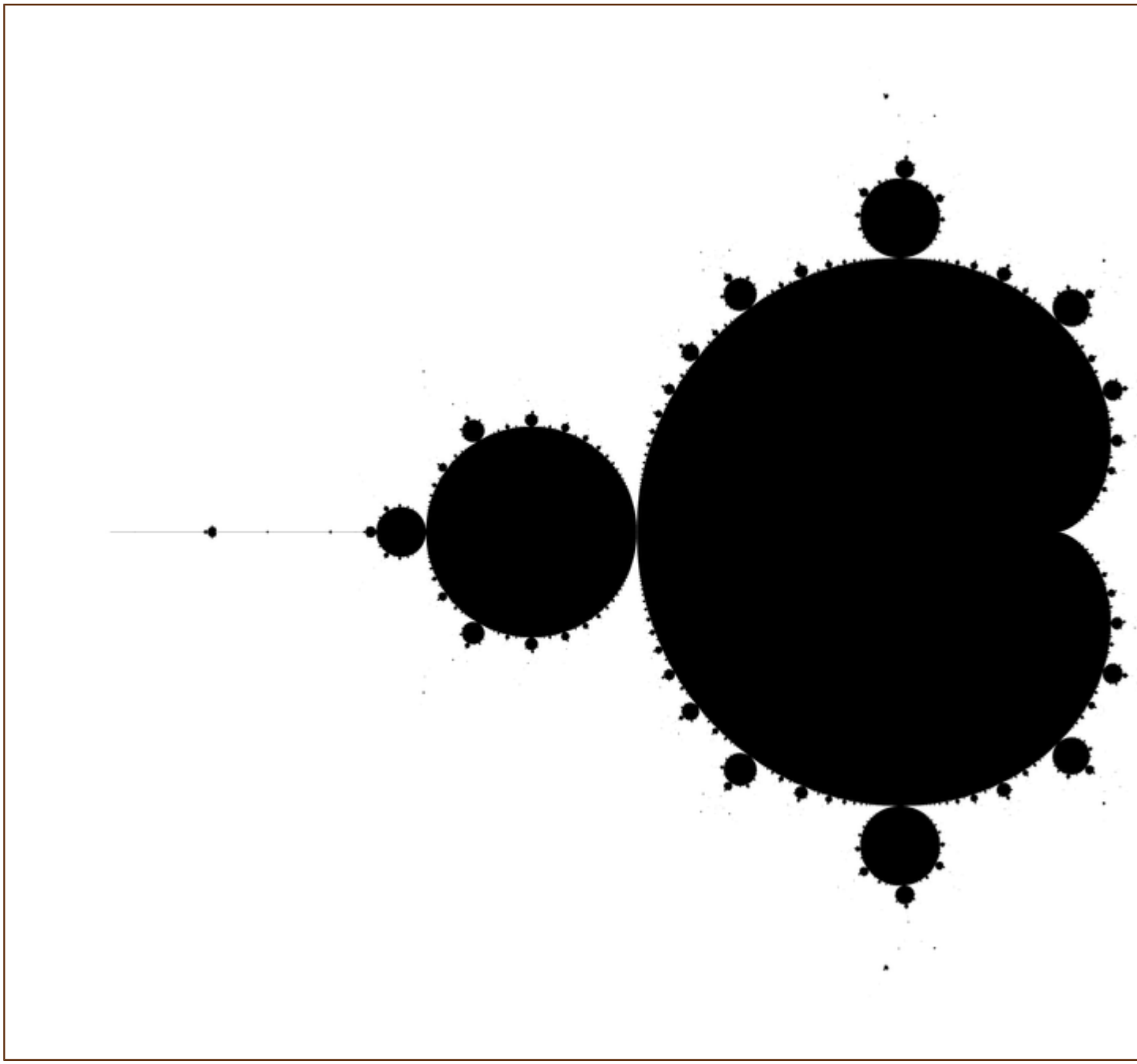


N=10^6

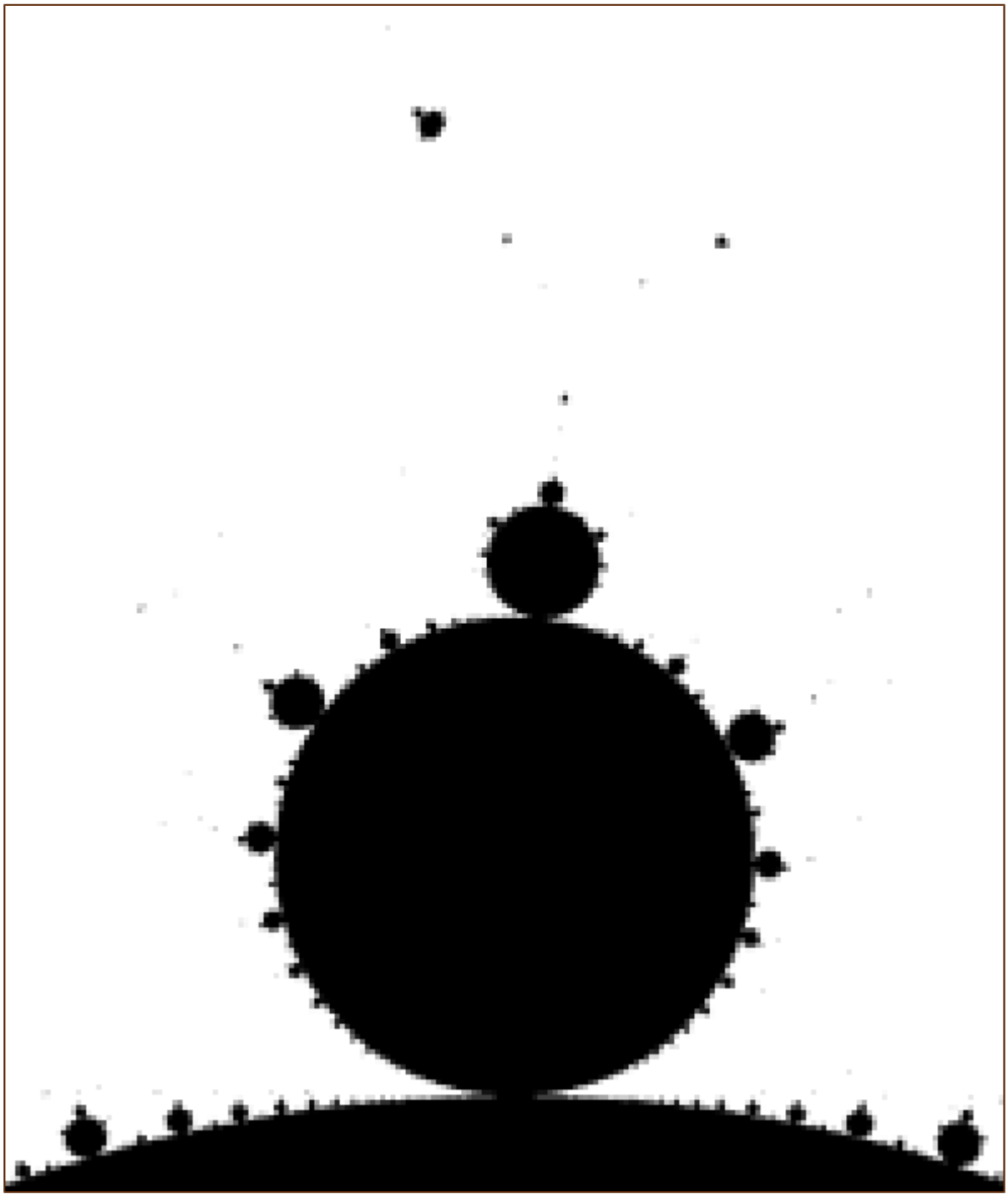
Let us look at a parts of a higher resolution image, computed with a big value of N: notice the small islands. The set M is connected in fact, but the strands connecting the different black parts are invisible, except some horizontal line that appears because, by coincidence, we are testing there complex numbers that belong to the real line: $M \cap \mathbb{R} = [-2, 0.25]$M∩R=[−2,0.25].



The next image has been computed using a 4801×4001px image and then downscaling by a factor of 6 (in both directions) to get an antialiased grayscale image. It also has been cropped down a little bit, to fit in this column without further rescaling.



Last, an enlargment to see the pixels of the above image:



Escape time based coloring

There is a simple and surprisingly efficient modification of the above algorithm.

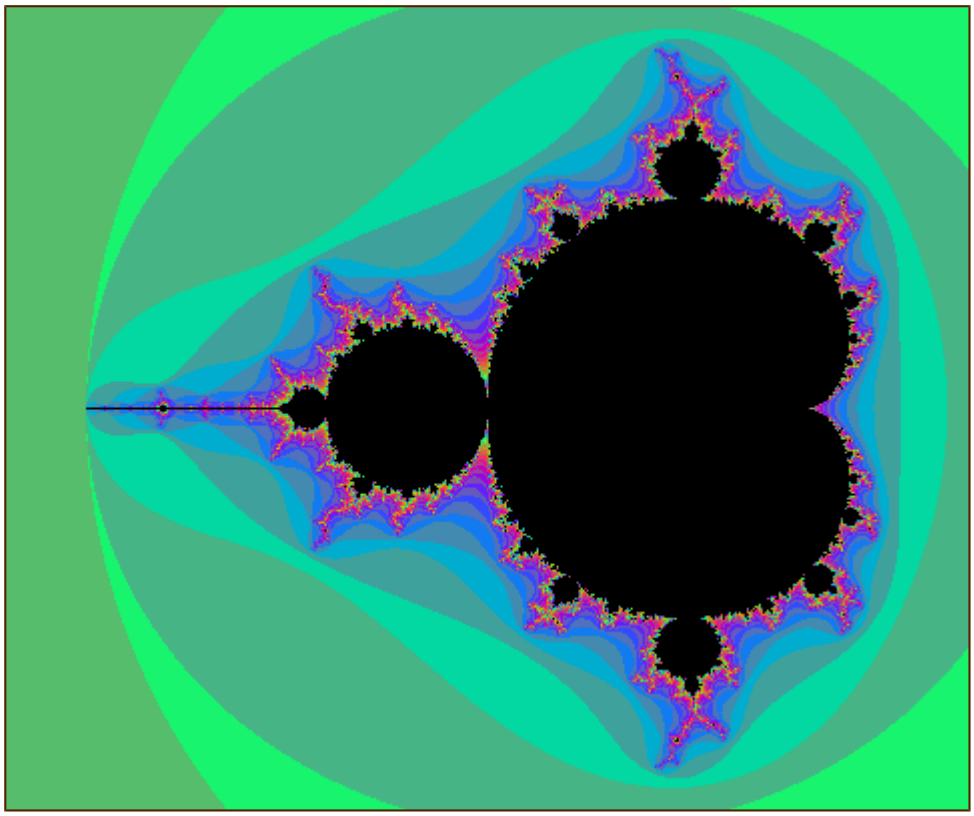
We started from z=0 and iterated the substitution $z \mapsto z2+cz \mapsto z2+c$ until either |z|>2 (the orbit *escapes*) or the number of iterations became too big. Instead of coloring white the pixels for which the orbit escapes, we assign a color that depends on n, the first iteration number for which |z|>2.

```
def f(n): # this function returns a color depending on an integer n
  return ... # put here your custom code
for p in allpixels:
  c = p.affix
  z = 0j
  color = black
  for n in range(0,N):
    if squared_modulus(z)>4:
      color = f(n)
      break
    z = z*z+c
```

```
  p.color = color
```

For the coloring, you are free to take your preferred function of n, I will not develop on that here.

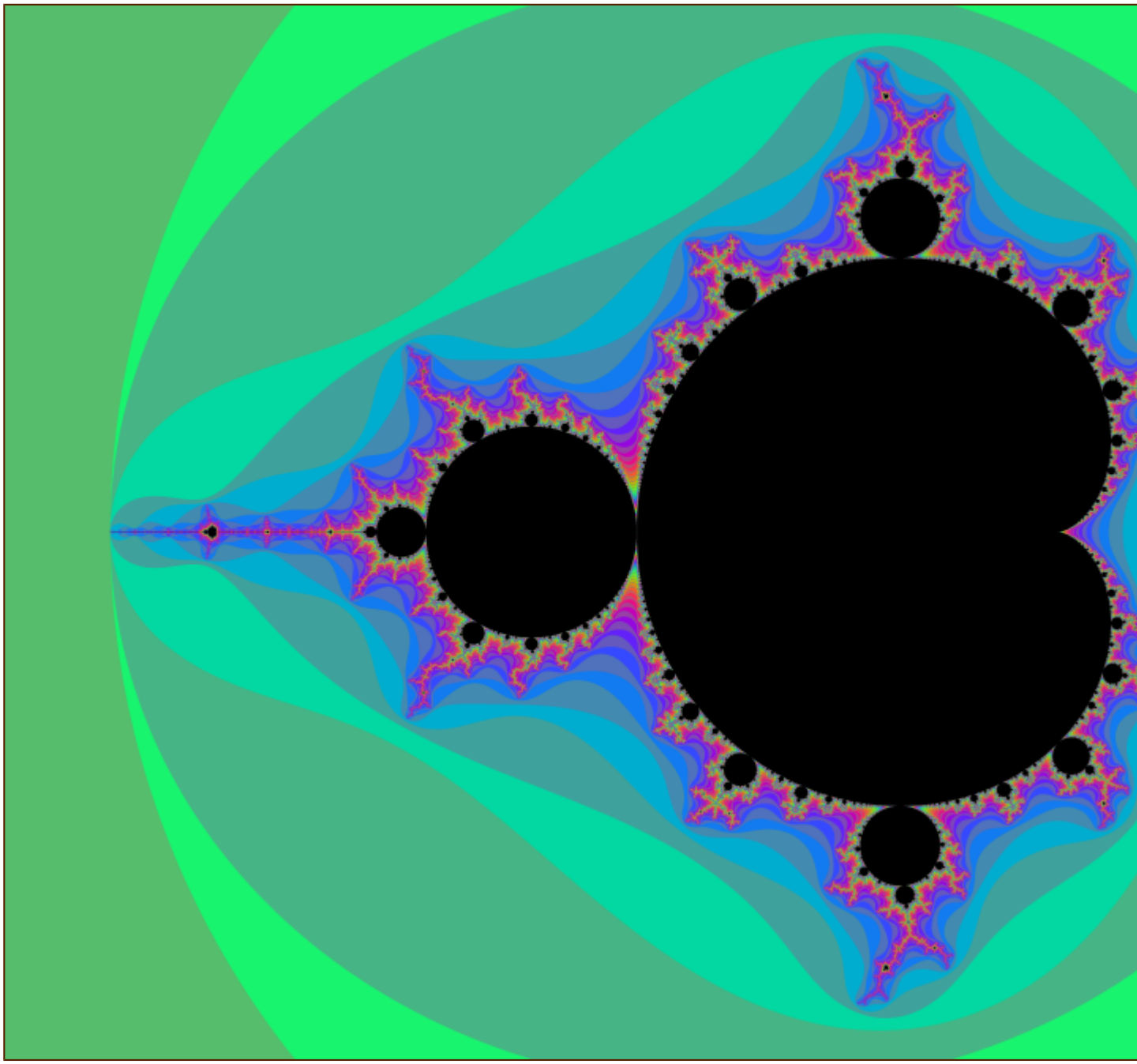
Here is an example of result:



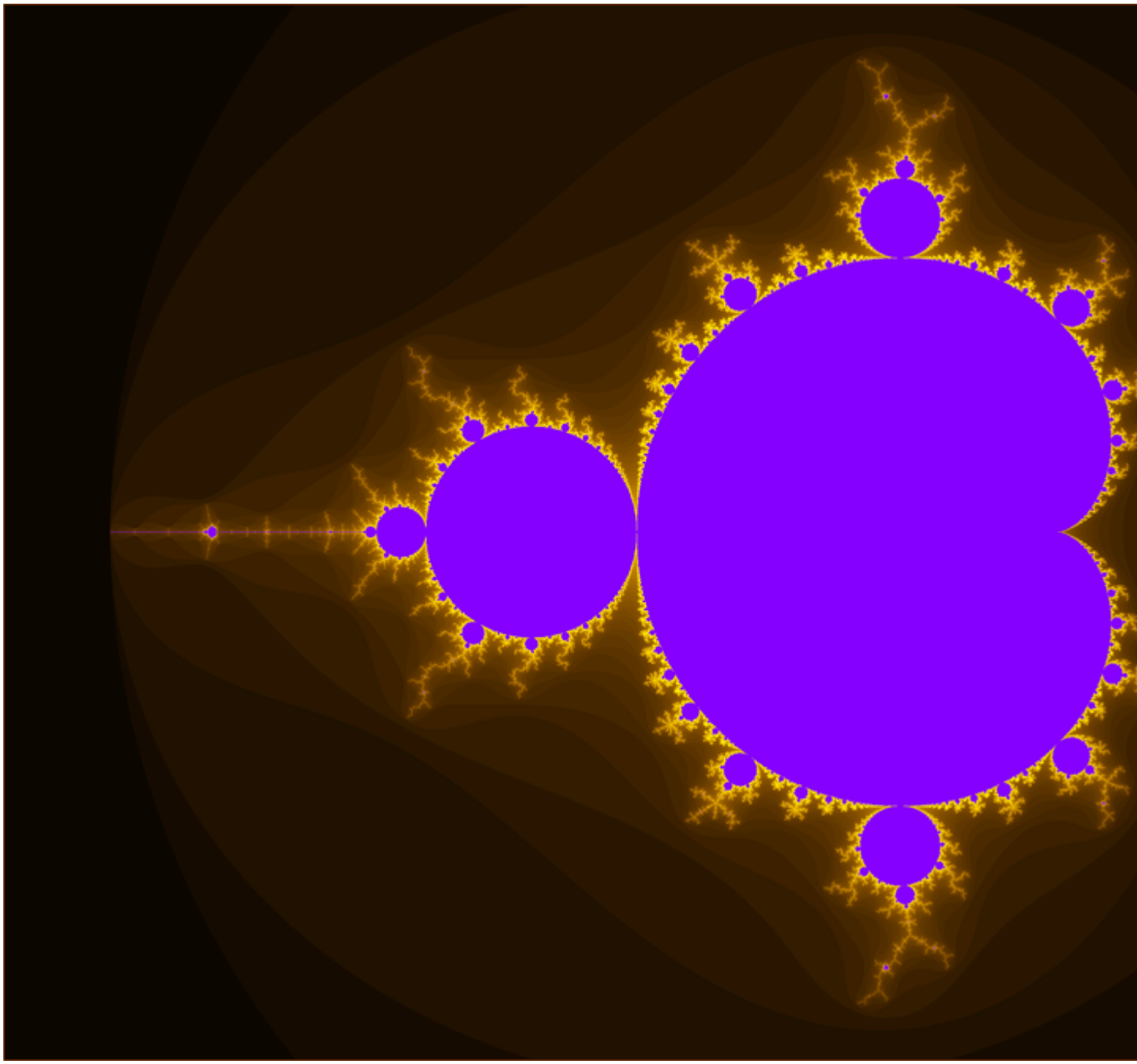
max iterations: 100, size: 481×401px

The strands are now visible.

Here is another one, on a 4801×4001px grid, downscaled by a factor 5, with N=10000 (which is probably a bit excessive in this particular case).

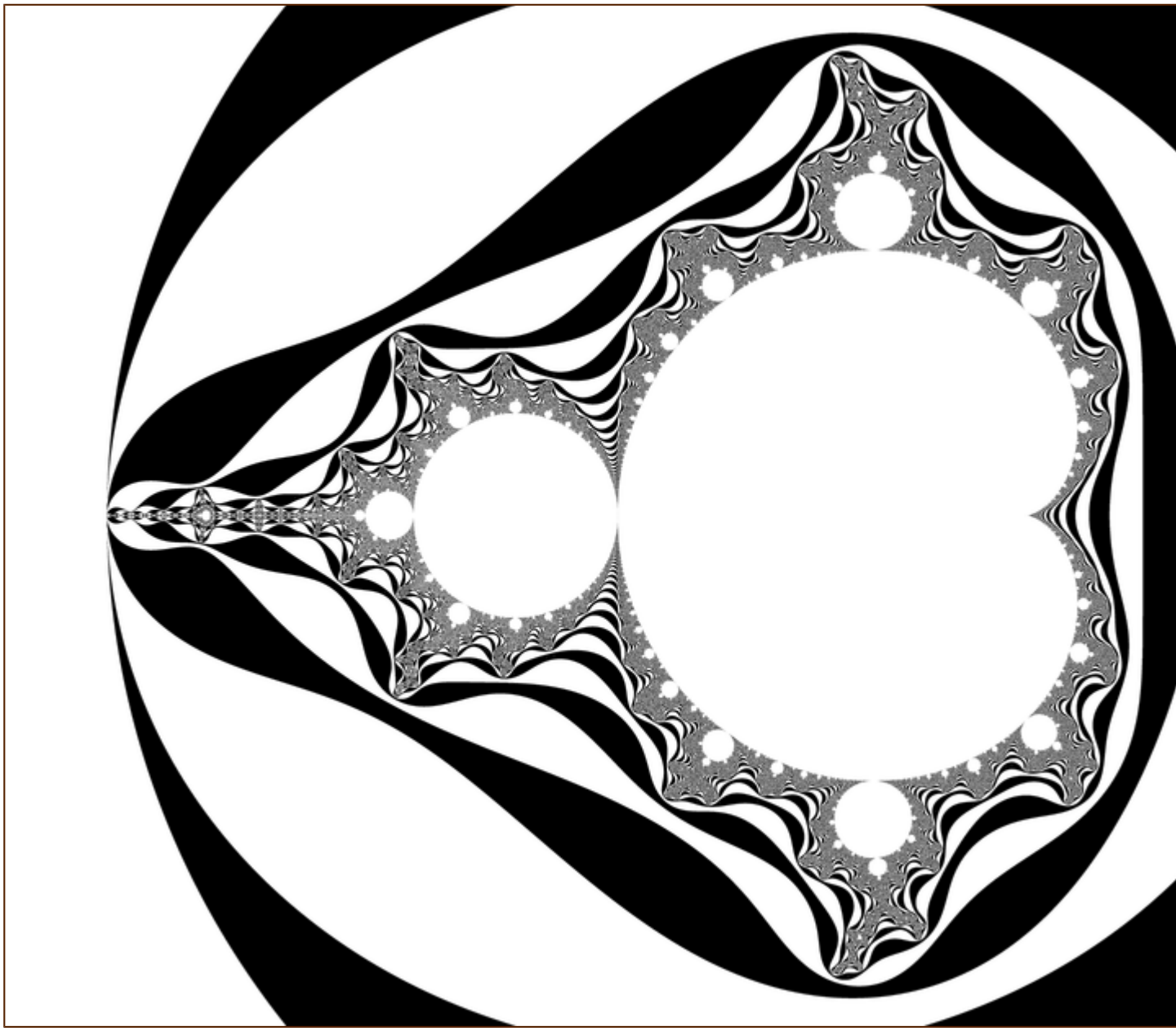


With a nicely chosen color function, one can get pretty results:



click for a higher resolution version

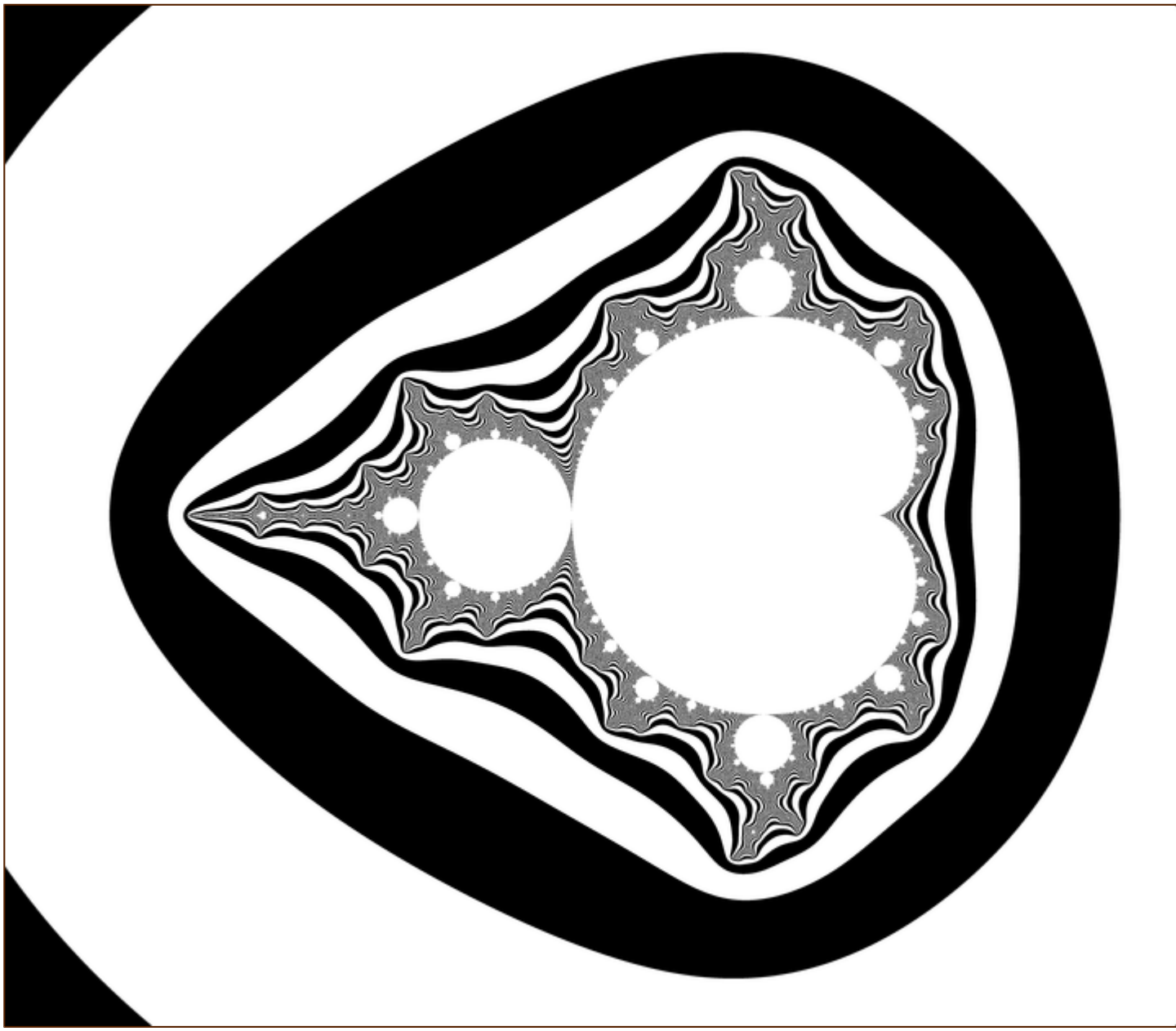
The following was popular in printed books:



click for a higher resolution version (still down-sampled from the original, though)

The potential

You may replace the test |z|>2 by |z|>R, provided that R>2. In fact the picture looks better that way. It has an explanation in terms of the *potential*.



R=1000, image downsampled by a factor of 8 in linear color space

> **Physics digression:** In very informal terms, in a 2D euclidean universe, put an electrostatic charge on some conducting bounded connected set K, the rest of 2D space being void (dielectric). You will put charge=-1 and assume there are so many particles (electrons) that the charge can be considered as a continuum. Then let the charges spontaneously move to minimize the energy. The resulting distribution has the property that the potential is constant on K. The potential function VV satisfies the *Laplace equation* $\Delta V=0$ΔV=0 outside K and $V(c)\sim\rho\log|c|$V(c)∼ρlog|c| when $|c|\longrightarrow+\infty$|c|⟶+∞, for some constant ρρ.

There is however no need to invoke electrostatics and one can define directly the following function and decide to call it **the potential**:

$$V(c)=\lim n\to+\infty\log+|Pnc(0)|2nV(c)=limn→+∞log+|Pcn(0)|2n$$

where $\log+(x)$log+(x) is defined as 00 if $x<1$x<1 and $\log(x)$log(x) otherwise. The exponent n in PncPcn refers to *composition*, not exponentiation: $Pn(z)$Pn(z) is P applied n times to z. The function V takes value 0 exactly on M. If x>0, the sets of equation V=x are called *equipotentials* of M. They foliate the outside of M into smooth simple closed curves that encircle it. The potential also finds an importance in the mathematical study of M, as holomorphic functions are involved, but that is not a the topic to be discussed here.

The fact that the formula converges and that its limit has the stated properties is not supposed to be obvious: it is a theorem too. One good thing about the formula is that it converges quite well: if we stop the iteration when |z|>R, the relative error will be of order 1/R. Now assume the picture is drawn for $|c|<10$|c|<10 (anyway all points of M satisfy $|c|\leq 2$|c|≤2) and at some point in the iteration of 00 we reach $|z|>4$|z|>4. Then a value of $|z|>1000$|z|>1000 is reached pretty fast because $|P(z)|=|z2+c|>|z|2-5$|P(z)|=|z2+c|>|z|2−5: one sees that in 4 more iterations we have at least |z|>443556...

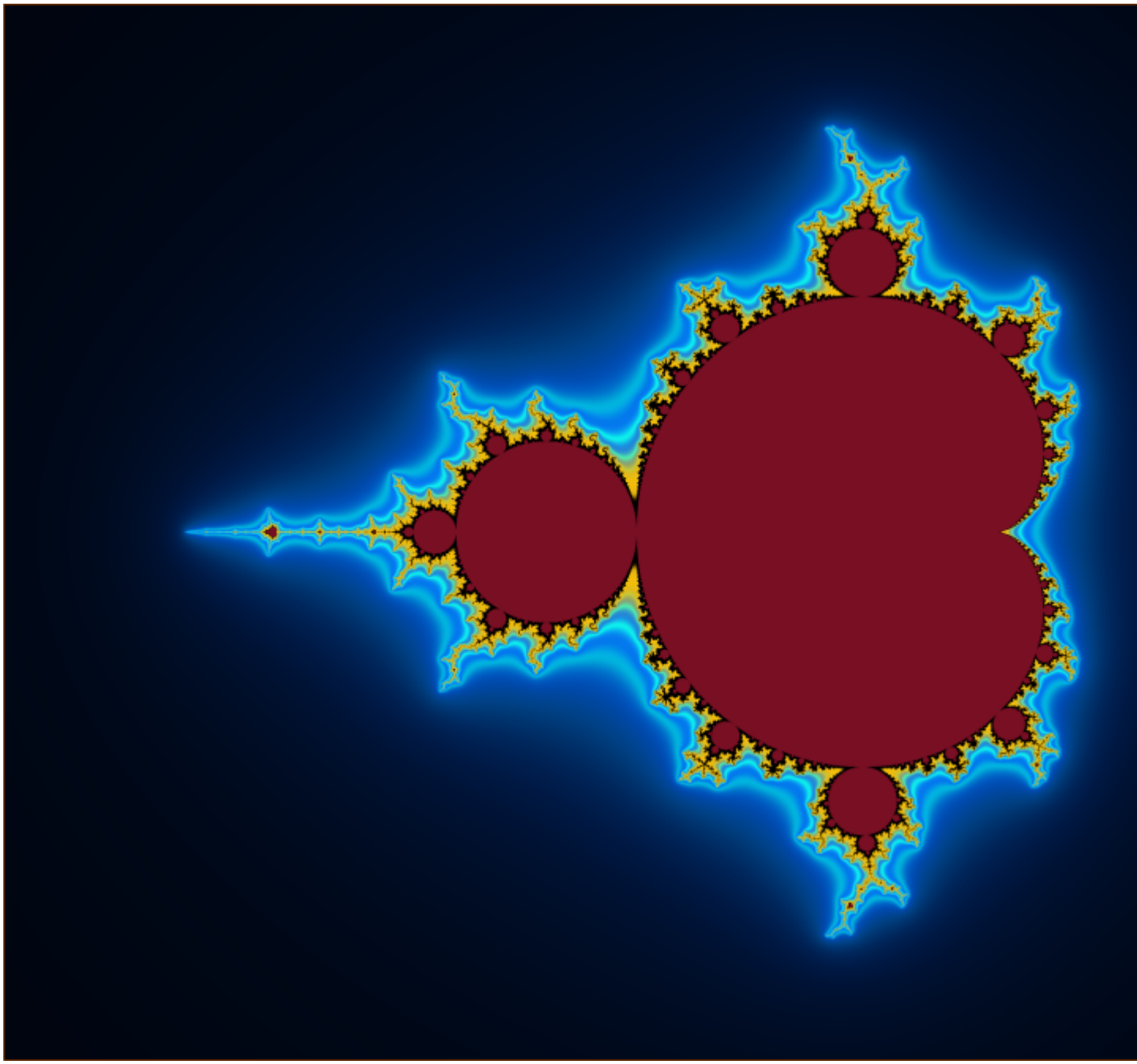
In the computation, when |z|=R after n iteration, and R is big enough, it means $V(c)\approx\log(R)/2^n$V(c)≈log(R)/2n. Hence taking R=1000 in the previous algorithm (see the picture above) yields color regions whose boundaries very closely match the equipotentials $V(c)=\log(1000)/2^n$V(c)=log(1000)/2n. This is what the picture above shows.

But we can do better: as soon as $|z|>1000$|z|>1000, we have a good approximation $V(c)\approx\log|z|/2^n$V(c)≈log|z|/2n, and we can thus choose a continuous coloring scheme depending on $V(c)$V(c).

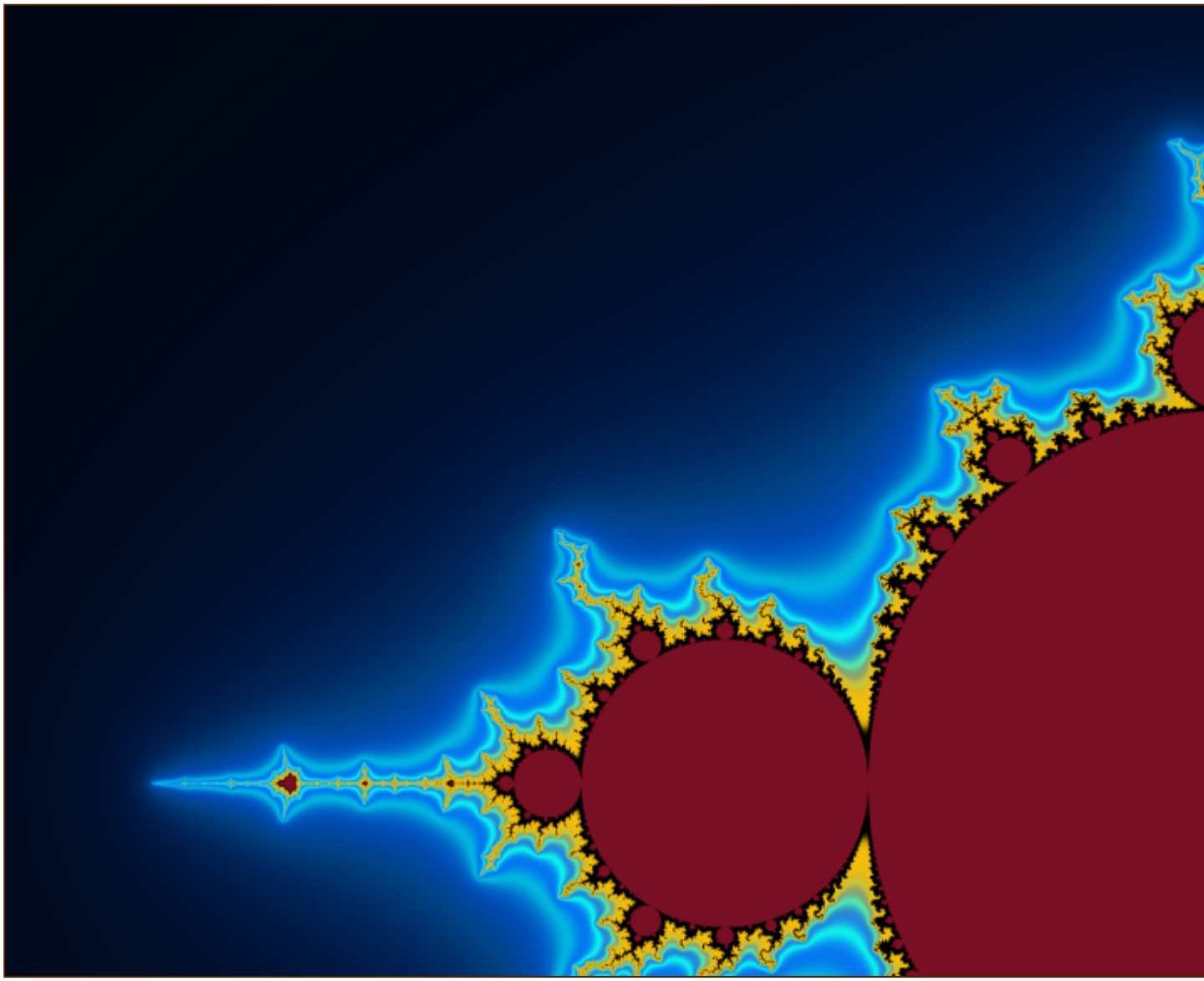
Here is a possible algorithm:

```
def f(V): # this function returns a color depending on a float V
  return ... # put here your custom code
for p in allpixels:
  c = p.affix
  z = 0j
  color = inside_color # color may be modified below and assigned to p
  pow = 1. # pow will hold 2^n
  for n in range(0,N):
    R2 = squared_modulus(z)
    if R2>1000000: # 1000 squared
      V = log(R2)/pow
      color = f(V)
      break # this will break the innermost for loop and jump just after
    z = z*z+c
    pow = pow * 2.
  p.color = color
```

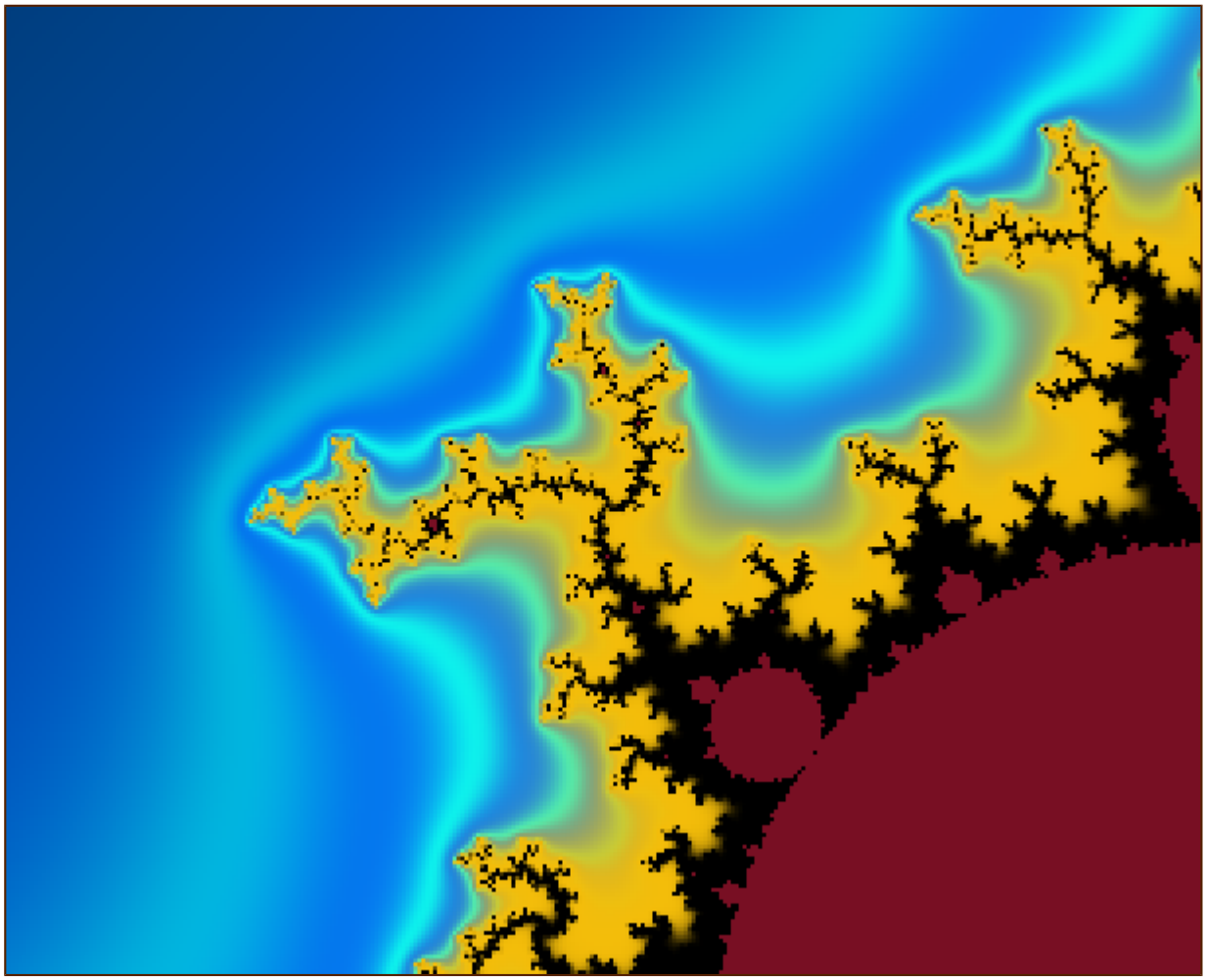
And here an example of result: computed with N=2000 and 4800×4000px and a color function that starts black away from M, then slighlty oscillate in shades of blue like a sine function of log(V), but when V gets small a uniform egg yellow color takes over, and even closer it is a black one. The set M is in dark bordeaux red.



Whole object, downsampled to 800x677 (factor 6)



downsampled by factor 4 and cropped



enlargment on a small portion of original image, no downsampling

Deep zooms and log-potential scale

The function VV is continuous and has value 00 exactly on M.

The function log$V$logV is better suited than VV for coloring deep zooms. For instance in many points (especially tips and branch points which are asymptotically self-similar [Tan Lei]) a zoom will essentially shift the value of log$V$logV by a adding a constant that depends on the point on which you zoom.

A simple choice is to have a color that cycles when some chosen constant is added to log$V$logV. This choice is also a heritage from when images had a limited palette. So choose K>0K>0 and set

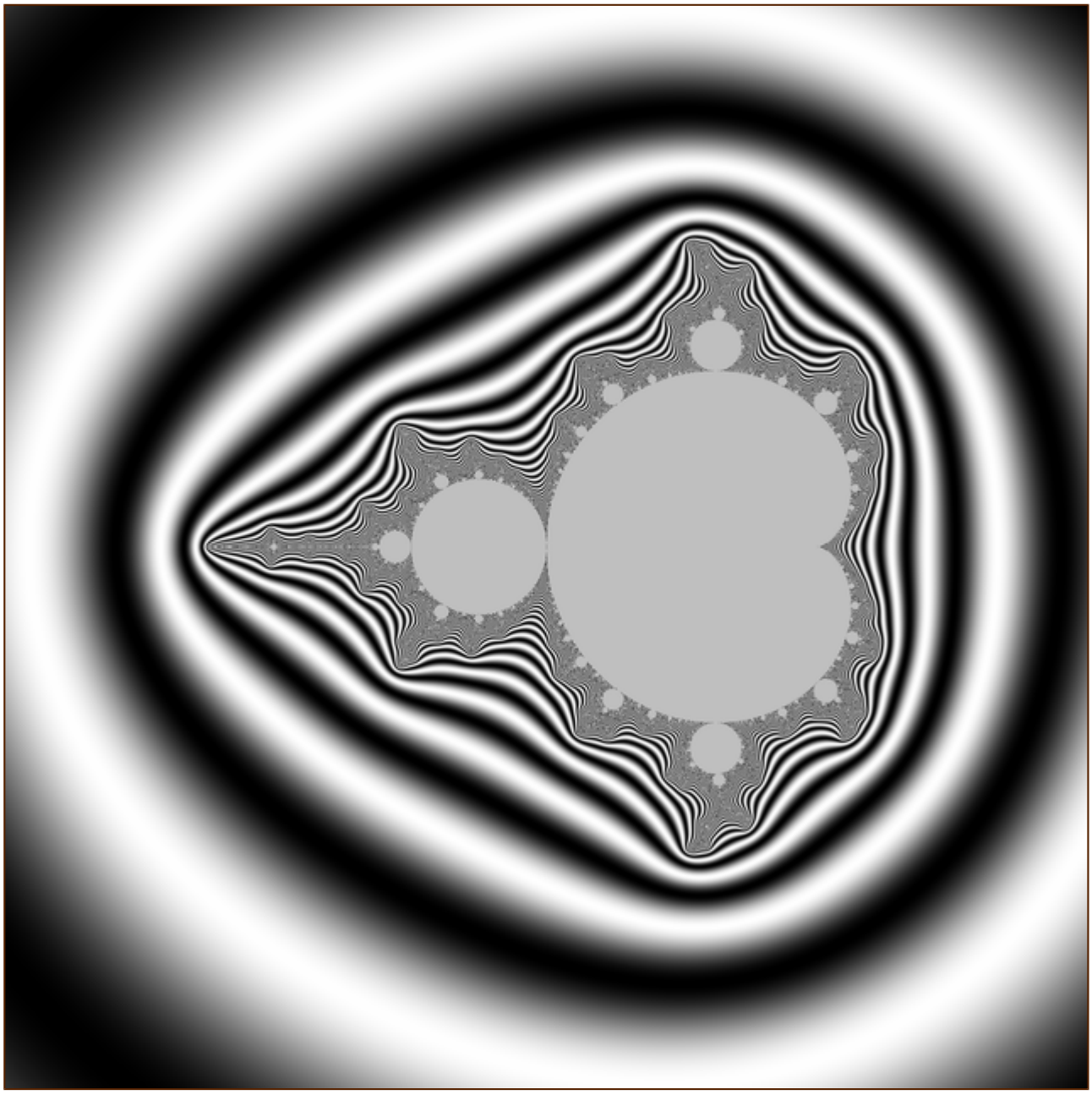
$$x=\log(V)/Kx=\log(V)/K$$

$$\text{color}=g(x)\text{color}=g(x)$$

for some function g with g(x+1)=g(x). You can enforce periodicity by reducing x modulo 1 to [0,1[but that may introduce discontinuities if g was not 1-periodic. For instance

$$g(x)=(R,G,B)=(255,255,255)\times 1+\cos(2\pi x)2g(x)=(R,G,B)=(255,255,255)\times 1+\cos(2\pi x)2$$

will yield a smooth wave pattern, from white to black and back.



K=log 2

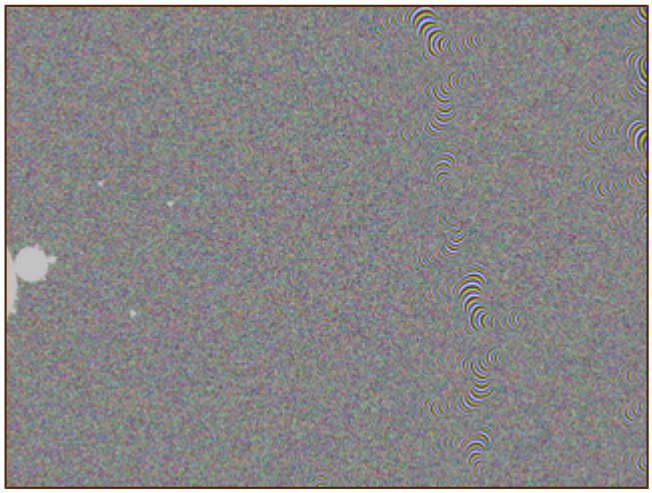
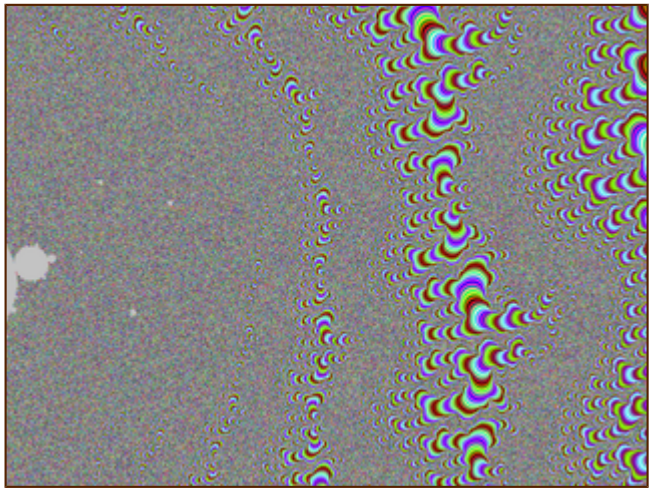
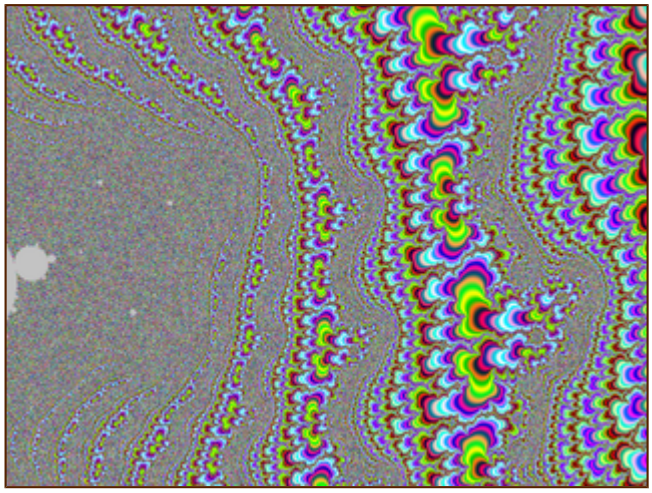
Taking K=log2K=log2 is a good starting choice, as it is "natural" for reasons we will explain later. You may already notice that loglog$P(z)$≈(loglog|z|)+log2loglogP(z)≈(loglog|z|)+log2 when z is big. However, M gets quite furry in some deep zooms, it is then better to divide by a bigger constant in this case.

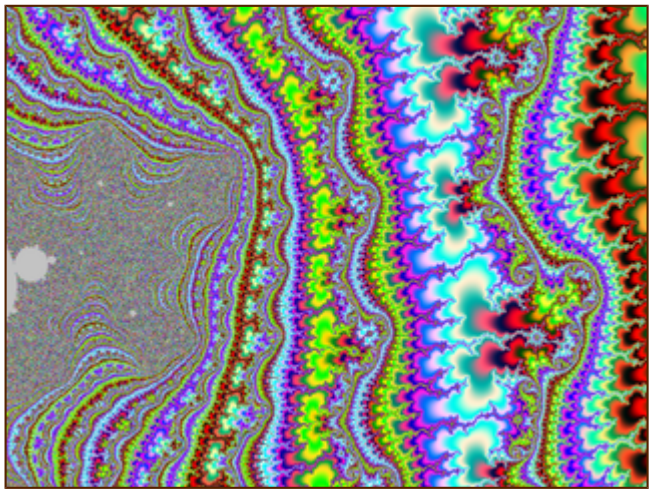
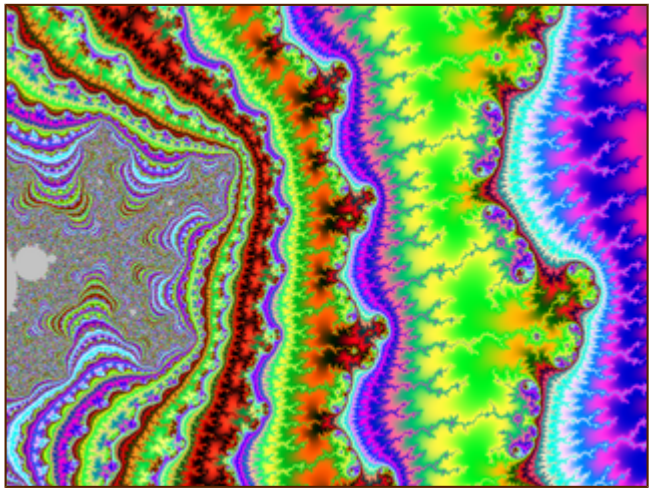
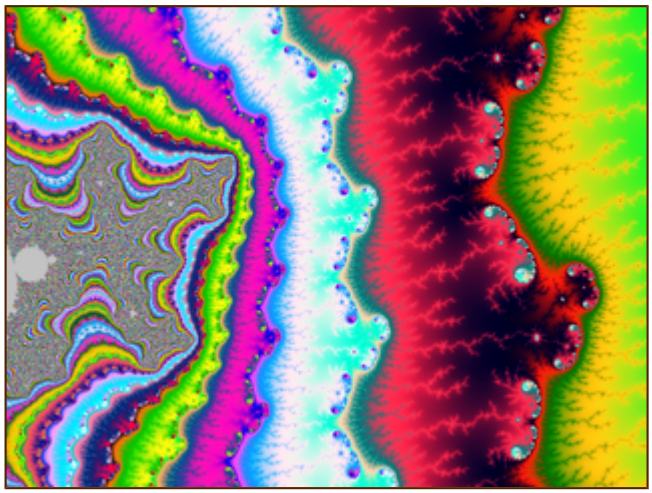
Below, some illustrations: Instead of a periodic function we chose a quasiperiodic one:

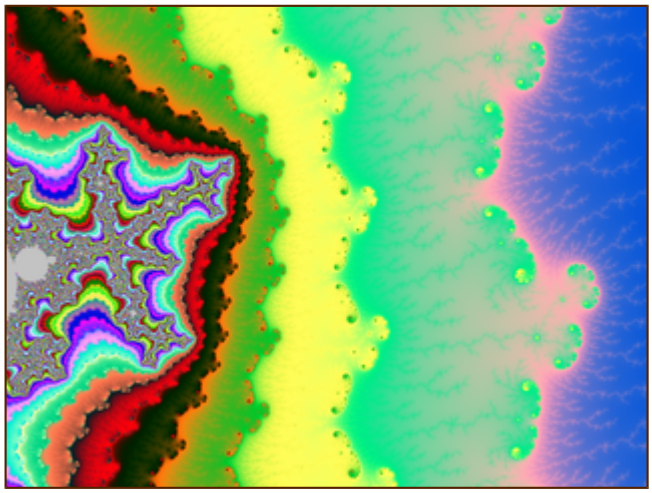
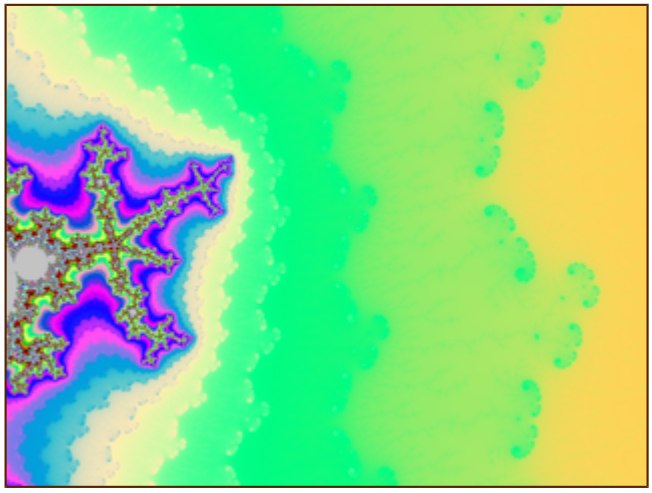
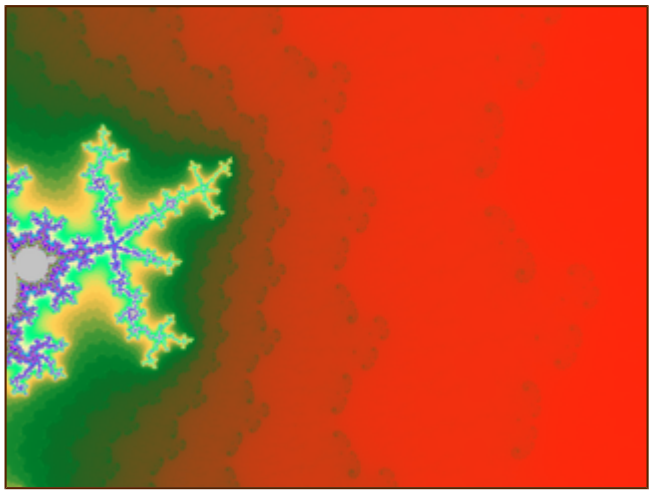
$$g(x)=(R,G,B)=\left(255\times 1-\cos(ax)2,255\times 1-\cos(bx)2,255\times 1-\cos(cx)2\right)\text{g(x)=(R,G,B)=(255×1−cos(ax)2,255×1−cos(bx)2,255×1−cos(cx)2)}$$

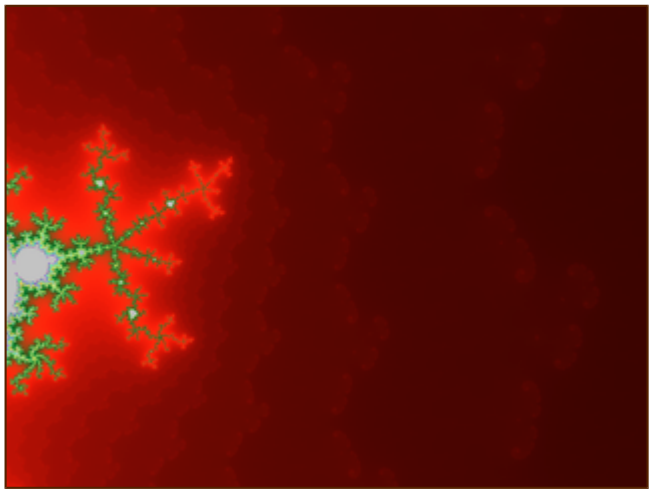
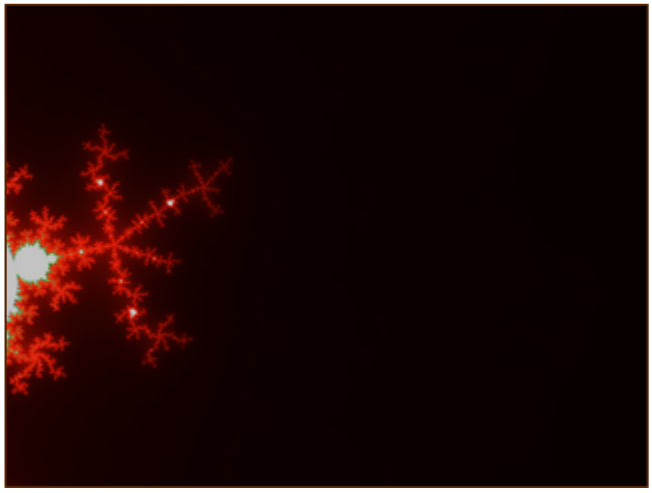
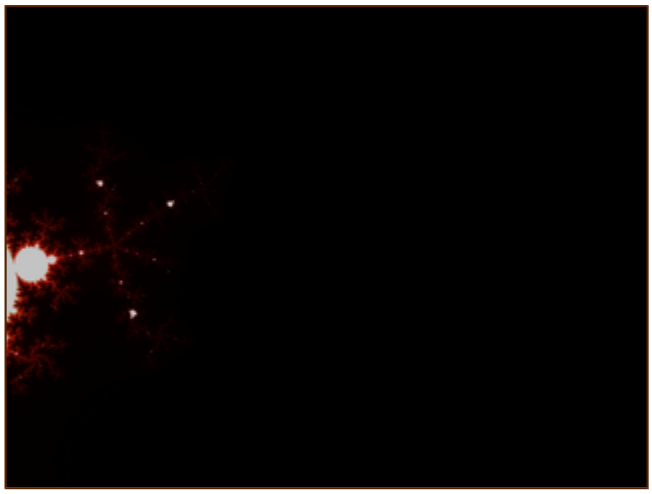
with $(a,b,c)=\left(1,132\sqrt{},17\cdot 31/8\right)\times 1\text{log2(a,b,c)=(1,132,17·31/8)×1log2}$ linearly independent over ℤZ. (It is not the prettiest choice; this is not the point, though.)

Below, 12 views of the same deep zoom on M. Only K varies, taking respective values 0.03, 0.1, 0.3, 1, 3, 10, 30, 100, 300, 1000, 3000 and 10000. They are all downscaled by a factor 2 from an original of 2400×1800px. They is a further rescaling in the thumbnails below, but you can click on them to get the 1200×900px version.

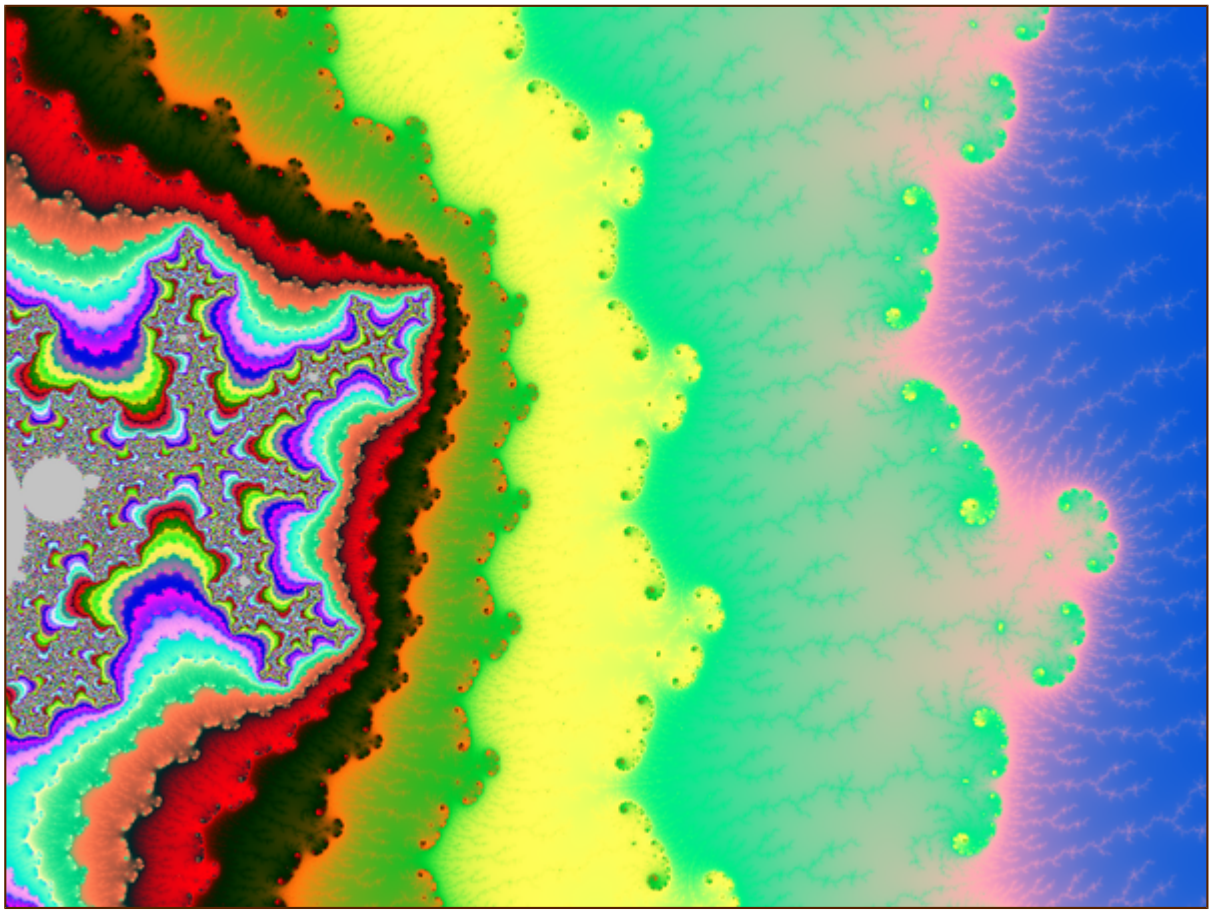
- 
- 
- 

- 
- 
- 

- 
- 
- 

- 
- 
- 

It is particularly visible on the 7th one, that not one value of K can capture all the filamental structure : on the right it is barely visible, and on the left near the circular gray shapes it is a mess.



Note: the computation of the images above have been accelerated using the interior detection method described later in this page.

Interior detection methods

Math digression: hyperbolic components

We must now introduce *hyperbolic components*. If you take c inside M, there is a good chance that c will belong to one.

Definitions:

- A *periodic point* for P is a point *a*a such that *Pk*(*a*)=*a*Pk(a)=a for some k>0.
- The first *k*>0k>0 for which this occurs is called the *period* of a.
- The orbit of *a*a is then finite: it is called a *cycle*.
- The value of (*Pk*)′(Pk)′ is the same at any point in the orbit of a: it is called the *multiplier* of the cycle.
- A cycle is called *attracting* or *attractive* or *a sink* if the multiplier has modulus<1.
- A parameter c is called *hyperbolic* if *Pc*Pc has an *attracting* cycle.

Recall that *Pc*Pc is defined by *Pc*(*z*)=*z*2+*c*Pc(z)=z2+c. For hyperbolic parameters, Fatou proved* that the critical point is attracted to this cycle. In particular, *c*∈*M*c∈M.

(*) This is a quite striking feature of non-locality holomorphic dynamics: the cycle has an attracting power that reaches to the critical point.

Fatou also prove that *Pc*Pc can have at most one attracting cycle.

Why call this "hyperbolic"? In more general dynamical systems, this word has another definition, which in our case would read as follows: *Pc*Pc is expanding on its Julia set. It turns out that the two definitions are equivalent, a highly non-trivial fact.

Hyperbolic parameters are stable: all nearby parameters are also hyperbolic, the attracting cycle and the Julia set move continuously when cc varies in a small neighborhood. The *hyperbolic component* associated to cc is the connected component of the set of hyperbolic parameters that contains c. Such a component is bounded by an algebraic curve, smooth except at at most one point. This curve is composed of those parameters for which the multiplier of the cycle has modulus one. In fact for this family PcPc, the components are either like disks or like interior of cardioids. They have no holes.

It can be proved that these boundary curves are contained in the topological boundary of M: hence a hyperbolic component is also a connected component of the interior of M. We do not know if the converse holds: this is *Fatou's conjecture*, still open today:

Conjecture: *All connected components of the interior of M are hyperbolic.*

It is a big conjecture. Solve it and become famous.

Following the derivative

As we iterate z, we can look at the derivatives of P at z. In our case it is quite simple: $P'(z)=2z$P′(z)=2z. Multiplying all these numbers along an orbit yields the derivative at z of the composition PnPn. This multiplication can be carried on iteratively as we iterate z:

```
  [...]
  der = 1
  z = c # note that we start with c instead of 0, to avoid multiplying the
derivative by 0
  for n in range(0,N):
    new_z = z*z+c
    new_der = der*2*z
    z = new_z
    der = new_der
    [...]
```

If we started from the critical point there is a 0 at the beginning of the product, so $(Pn)'(0)=0$(Pn)′(0)=0. However, omitting the first term yields $(Pn)'(c)$(Pn)′(c), which has a good chance to be non-zero: it will be 0 if and only if 0 is periodic, which corresponds to the *centers of the hyperbolic components* of M, a notion that I will not develop here: you just need to know that they lie in the smooth areas inside M.

Note: See the lines?:

```
    new_z = z*z+c
    new_der = der*2*z
    z = new_z
    der = new_der
```

It is to avoid a frequent mistake:

```
    z = z*z+c
    der = der*2*z
```

Would not work, whereas

```
    der = der*2*z
    z = z*z+c
```

would. See why?

The idea

Now if cc is a hyperbolic parameter then its orbit tends to an attracting cycle and therefore the derivative above will tend to 00. The converse does not exactly hold, but the counterexample values of cc are sparse.

The basic idea is then to choose a threshold and stop the computation when the derivative $(Pn)'(c)$(Pn)′(c) has reached a modulus below this threshold $\epsilon\epsilon$ and declare the pixel to be in M (in its interior in fact) and color it according to this and your taste.

It may wrongly mark as interior some points that are not. The hope is that the mistake is barely noticeable.

```
for p in allpixels:
  c = p.affix
  z = c
  der = 1+0j
  color = not_enough_iterates_color # replace it by inside_color if you prefer
  for n in range(0,N):
    if squared_modulus(der) < eps*eps: # eps is the threshold, a small number
      color = inside_color
      break    # here, this jumps to the last line
    if squared_modulus(z) > R*R:   # put here your preferred R>=2
      color = outside_color        # replace here by your preferred color scheme
outside M
      break    # here, this jumps to the last line
    der = der*2*z   # order matters: do not modify z --before-- computing the new
der
    z = z*z+c       # order matters
  p.color=color # last line
```

This method was first explained to me by Xavier Buff, who tested it and told me it gives good result, which I never would have thought.

So, does criterion work? Does getting below $\epsilon\epsilon$ ensure that there is an attracting periodic cycle ? What would be the danger? The orbit of cc could spend too much time in the area where $|P'|<1$|P′|<1, or it may pass a few times too close to 00. In the second case it seems intuitive that cc must be close to the center of a hyperbolic component. But what about the first case?

In fact there are some well chosen non-hyperbolic values of cc such that $\liminf n(Pn)'(c)=0\liminf n(Pn)'(c)=0$, but these are very rare. Moreover, any point outside M will satisfy $\inf(Pn)'(c)\geq f(V(c))\inf(Pn)'(c)\geq f(V(c))$ for some function $f>0f>0$, thus at least the points detected by the method that are not in M are close to M, and this distance can be made small by decreasing the threshold.

Images and speed

In short:

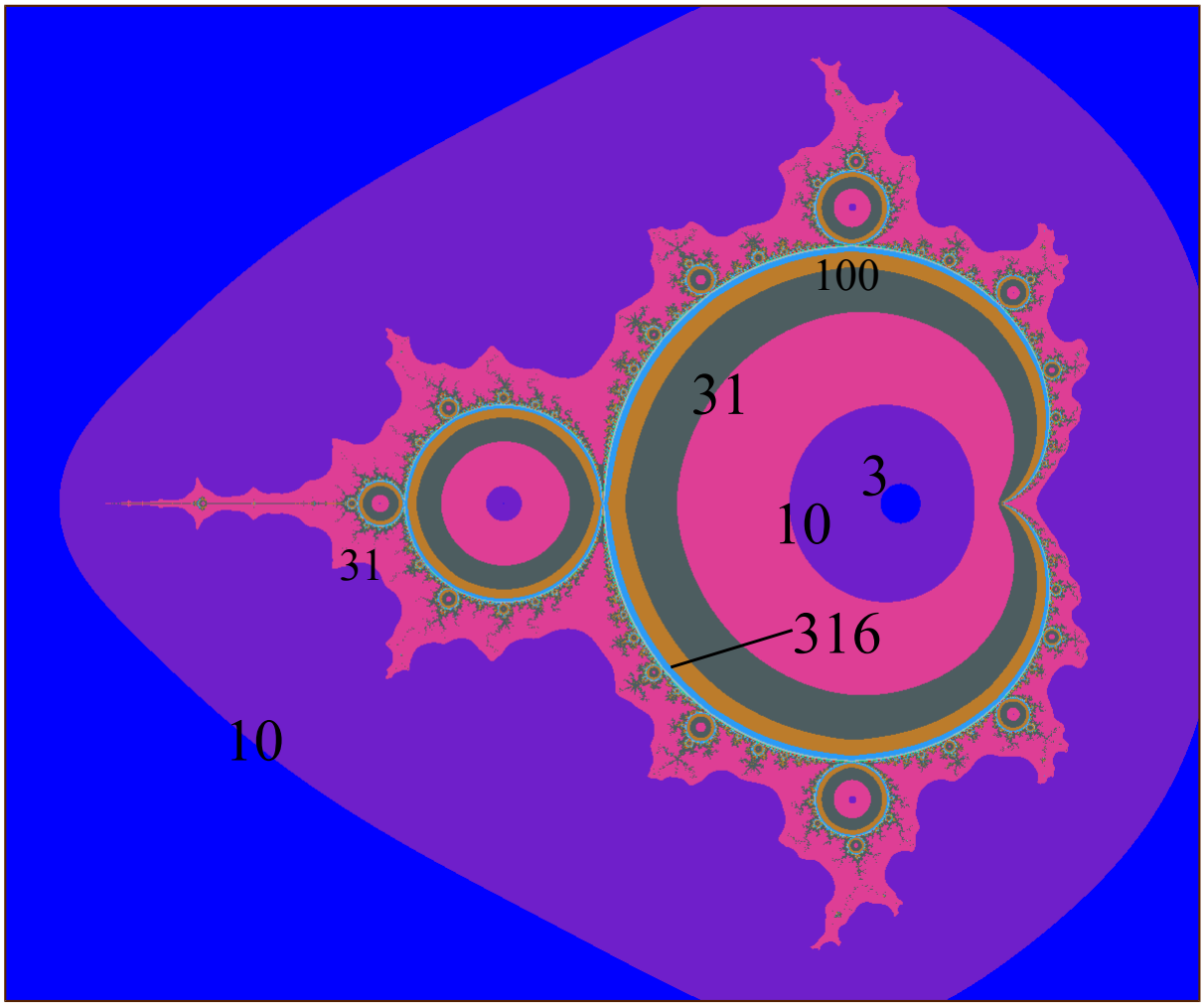
The improvement in speed is not so impressive in the case of M. However *the usefulness of the interior detection method appears elsewhere*: to get a comparable computation time without it, one has to fine-tune the maximal number of iterations N; when you zoom this has to be adapted by hand. With the method you can save this pain: you can start with a too big value, it makes little difference in the computation time.

Let us be more precise:

Notice that each iteration of the new method requires twice the amount of computations as an iteration with the old method (the ratio may be closer to 1 with more elaborate families and/or algorithms). Even if the interior of M covers a big part of the picture, there seems to be a well-chosen value of NN such that the average number of iterates necessary to reach a given accuracy with the old method is no more than 3 times than with the new method. So the computation time is only divided by 1.5.

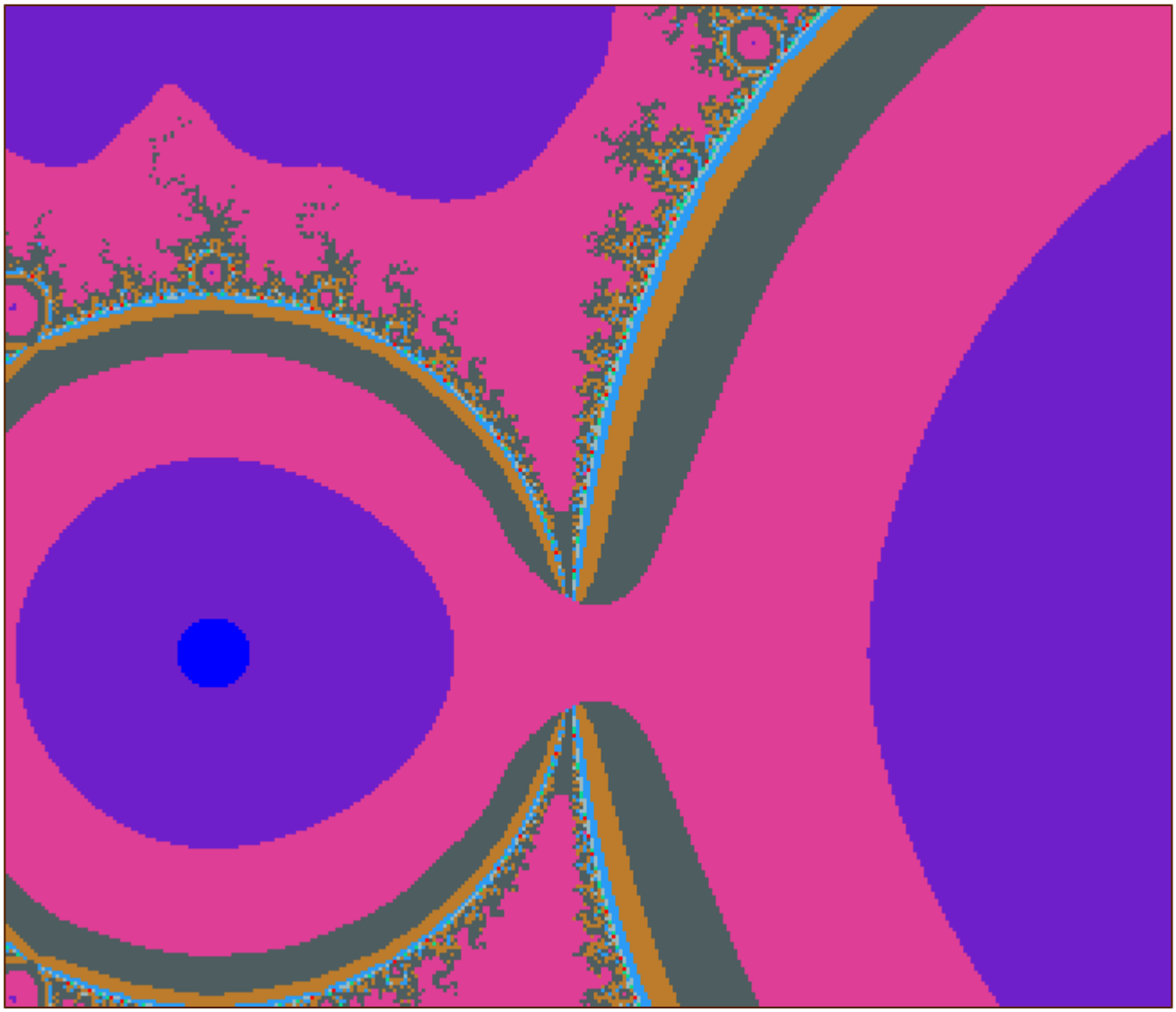
There is nevertheless another aspect for which the interior detection trick has a clear advantage. You can use a too big N even at low zoom factors (N=106N=106 is OK, but avoid N=10101010), it will barely make a difference because most points will stop much earlier anyway. This number N will work for a bigger range of zoom factors before you have to increase it.

Below, a simple experiment:



Color indicates the number of iterations, for instance, pink means between 10 and 31 iterations. Max iteration number N=10000. Inside computed with epsilon=0.001, outside with R=10.

It is quite surprising that a not-so-small value of epsilon gives a pretty good image of M. The problems seem to concentrate at parabolic parameters. Below a zoom near c=-3/4, with epsilon=1/10, enlarged to make pixels visible.



epsilon=0.1 makes the neck between the two big components slighlty wrong: the pink area should not be connecting them. This replicates in several other places.

Boundary detection methods via distance estimators

Milnor's

The first distance estimator method that I learned of was taught to me by Douady, it seems to be due to Milnor and Thurston, see [1],[2].

Principle

If 00 escapes under the iteration of $P:z\mapsto z2+c$P:z↦z2+c, let $z0=0$z0=0 and $\rho n=dzn/dc$ρn=dzn/dc. Choose a threshold $R>2$R>2 big enough (for instance $R=1000$R=1000). Choose an nn such that $|zn|>R$|zn|>R. An approximation of the distance from zz to MM is then given by the quantity

$$dn=2|zn|\log|zn|\rho n.$$dn=2|zn|log|zn|ρn.

The sequence ρnρn can be computed iteratively along with znzn: $\rho 0=0$ρ0=0 and

$$\rho n+1=2zn\rho n+1$$ρn+1=2znρn+1

Note: This is a bit analogous to the computation done in Section Following the derivative, with one important difference: we compute $dPnc(c)dc$dPcn(c)dc, *not* $dPnc(z)dz$dPcn(z)dz (where cc would be constant); the resulting inductive formula is thus different.

[insert example with formula wrong]

Caution: It is not true that this approximation $d_n$dn tends to the actual distance dd from zz to MM when nn tends to infinity. In fact, only weaker inequalities hold: $d_n$dn converges and $1/5<d/\lim d_n<1$1/5<d/limdn<1 provided $|z|<5.7$|z|<5.7. The lower bound 1/5 can be replaced by a function that depends on $|z|$|z|, but can never be better than 1/4. If $R=1000$R=1000 and $|z_n|>R$|zn|>R, then $d_n$dn is pretty close to $\lim d_n$limdn.

Math

The math behind the method has a higher level of sophistication. It is based on the following:

- A conformal map from the outside of MM to UU, the outside of the (closed) unit disk, is known: this is $c\mapsto\phi_c(c)$c↦ϕc(c), where ϕ_cϕc is the conformal map from the outside of the filled-in Julia set to UU.
- A simple and efficient way of estimating $|\phi_c(c)|$|ϕc(c)| and $|\partial\phi_c(c)/\partial c|$|∂ϕc(c)/∂c|.
- Conformal maps respect the hyperbolic metric.
- A correlation between the hyperbolic metric coefficient and the Euclidean distance to the boundary.

Let me give a few more comments on that, though I won't explain the whole thing. First, the infinitesimal expression of a hyperbolic metric takes the form $\rho(z)|dz|$ρ(z)|dz| where $\rho(z)>0$ρ(z)>0 (such expressions are called conformal metrics). Then, the hyperbolic metric on UU has the following infinitesimal expression: $|dz|/(2|z|\log|z|)$|dz|/(2|z|log|z|). Concerning the 3rd point, if ff is holomorphic, then the image of this metric by f^{-1}f−1 has expression $|f'(z)|\rho(f(z))|dz|$|f'(z)|ρ(f(z))|dz|. Concerning the 4th point above, if we were dealing with a simply connected set (which we aren't) then there is a very nice theorem : on a simply connected open subset OO of the complex plane, the coefficient ρρ of the hyperbolic metric $\rho(z)|dz|$ρ(z)|dz| lies between $1/d$1/d and $4/d$4/d where dd is the distance from zz to the boundary of OO. I think the idea in the method above is that it remains nearly true: other estimates, close to this one, can be given. I have not tried to figure out a nice and correct estimate so I'm not giving one here.

Implementation

Note: Don't forget in your program to: either modify ρ_nρn *before* $z_n$zn; or better, store the new values of ρρ and zz in temporary variables and only after assign them to ρρ and zz. The algorithm should look like that:

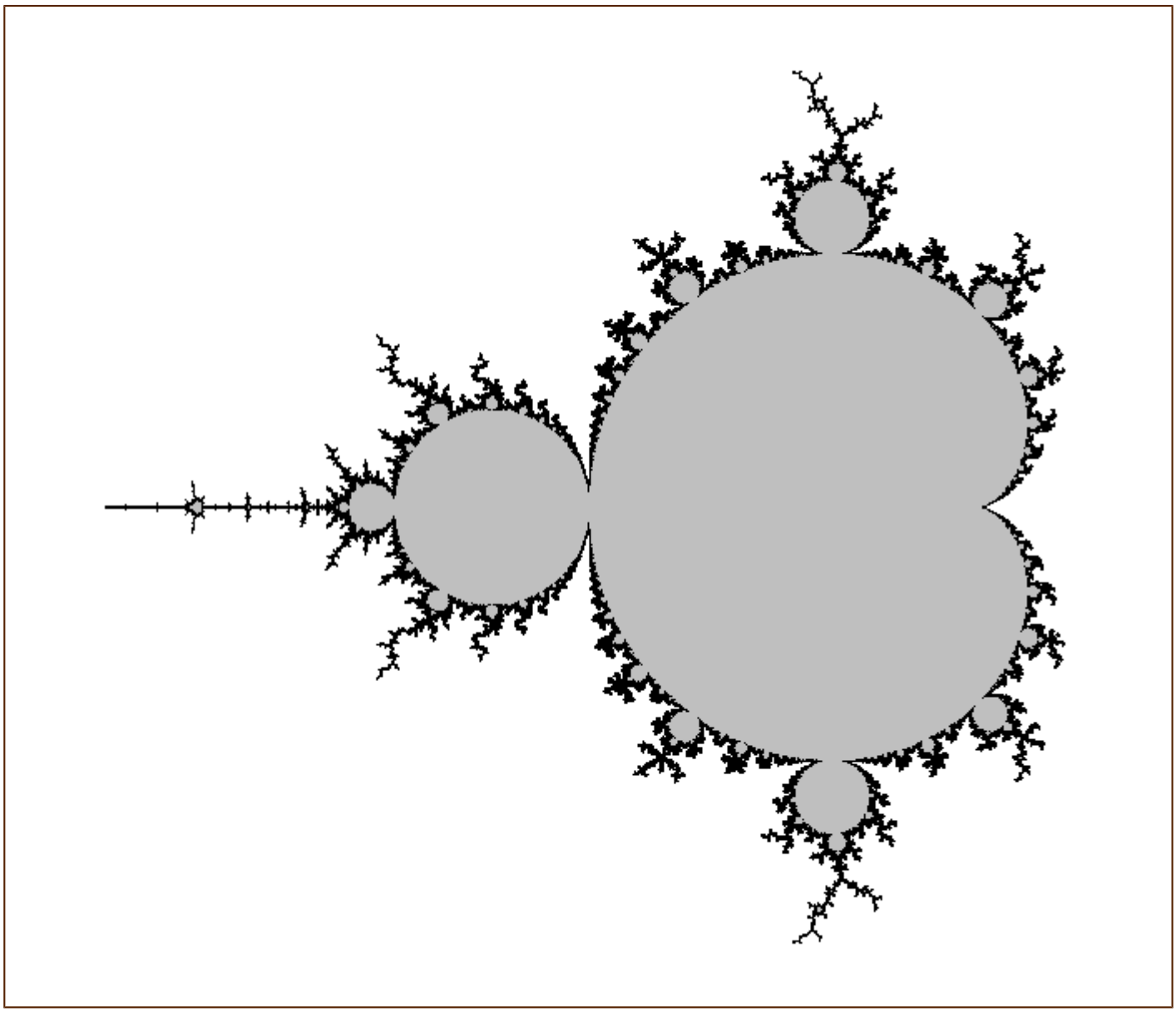
```
thickness_factor = 1. # change this if you want to tune the boundary thickness

for p in allpixels:
  c = p.affix
  z = c
  der_c = 1+0j # this is the derivative w.r.t c
  reason = NOT_ENOUGH_ITERATES

  for n in range(0,N):
    if squared_modulus(z) > R*R:
      reason = OUTSIDE
      break
    new_z = z*z+c
    new_der_c = der_c*2*z + 1   # notice the difference with the formula for the
z-derivative
    z = new_z
    der_c = new_der_c
  # the break above jumps here

  if reason == NOT_ENOUGH_ITERATES:
    p.color = not_enough_iterates_color
  else: # in this case reason = OUTSIDE
    rsq = squared_modulus(z)
    # the test below is equivalent to testing d_n < thickness_factor*pixel_size
    # d_n is defined in the text above this code snippet
    # x**y means x to the power y
```

```
    if rsq*(log(rsq)**2) < squared_modulus(thickness_factor*pixel_size*der_c) :
      p.color = boundary_color
    else:
      p.color = outside_color
```

Note that our test d_n < thickness_factor*pixel_size is done at the level of squares, as a mild way to save computation time. It can be further optimized. However, it does not save much because most of the time is spent in the inner loop.

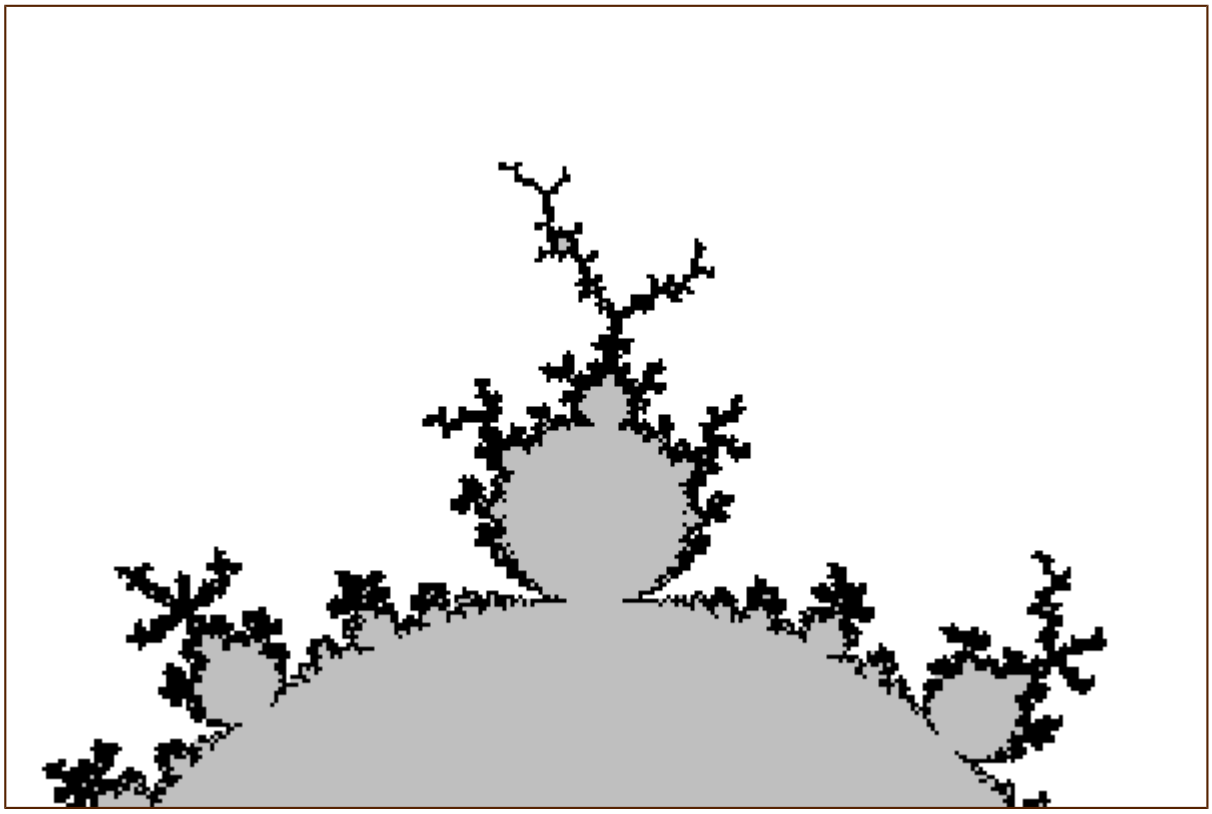
Tests

We start with the whole beast:

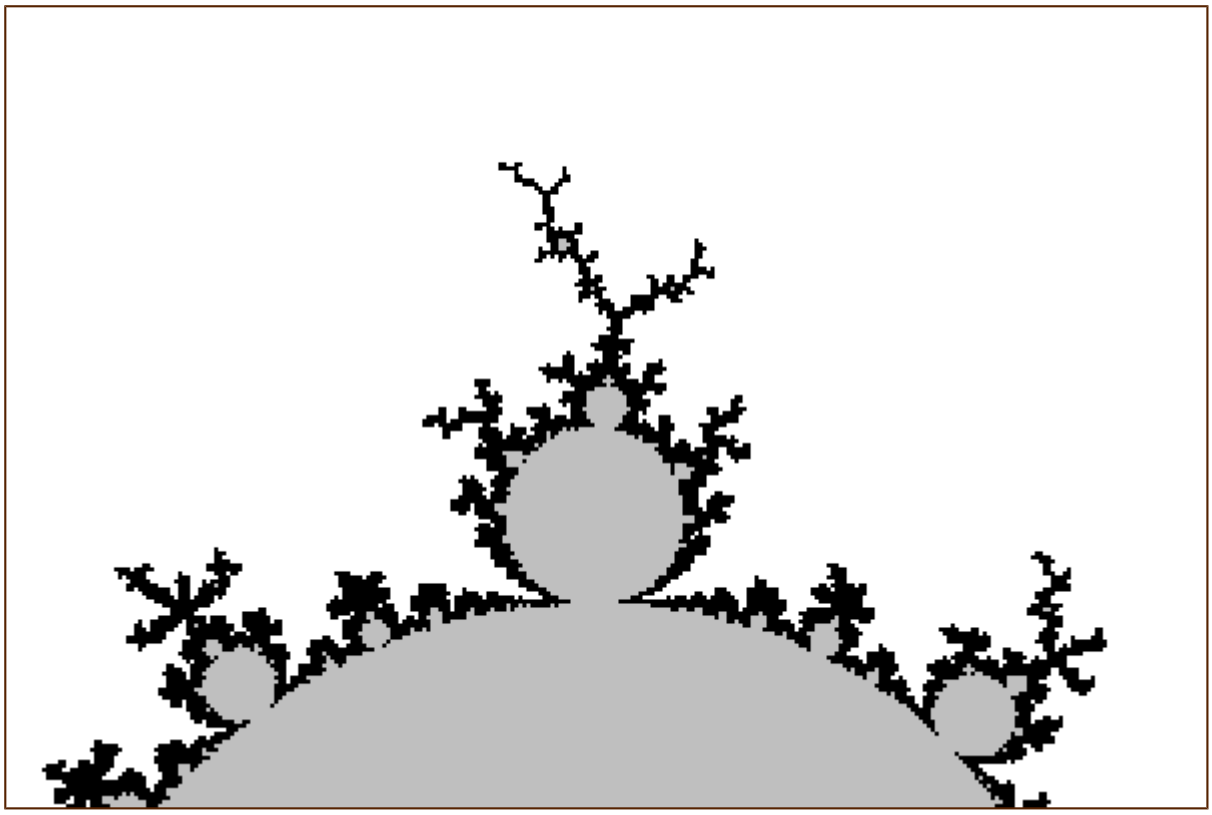


max iterations = 1000

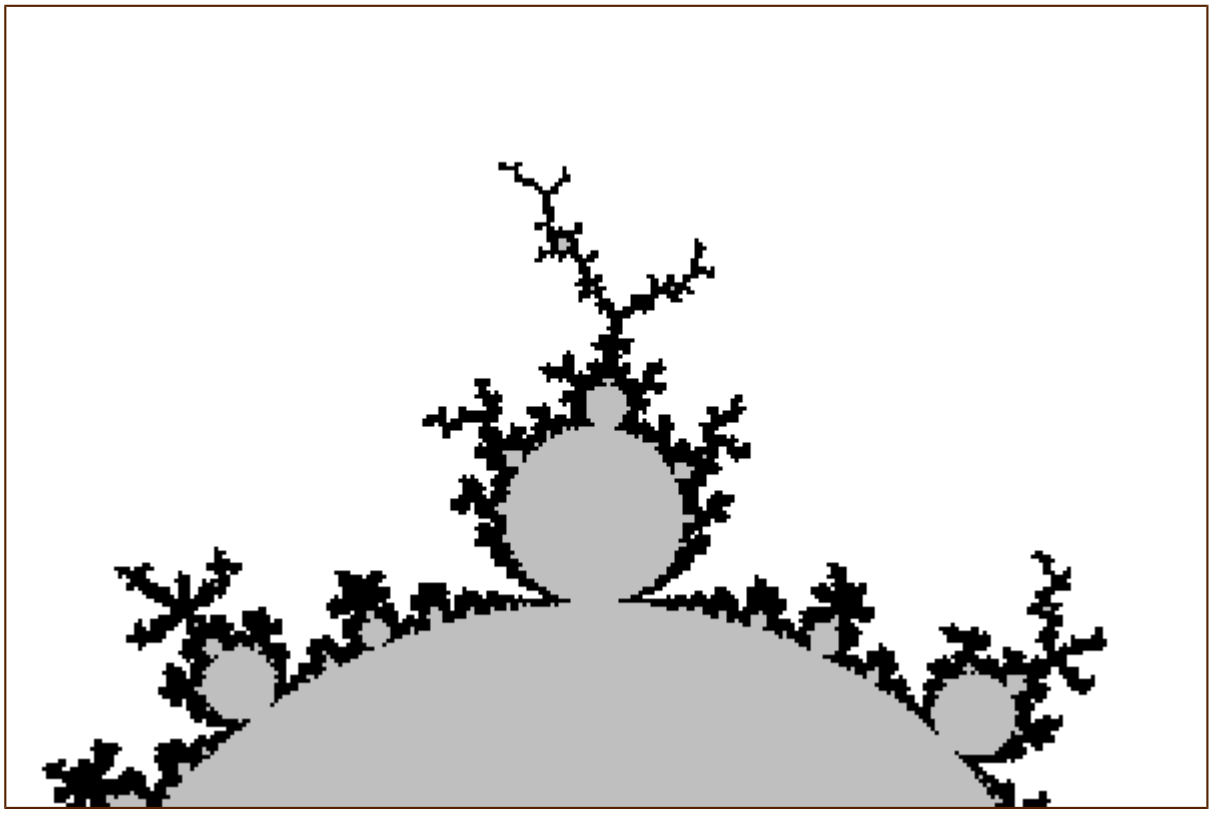
Comparison with different values of the maximal iteration number (zoomed) :



max iter = 100

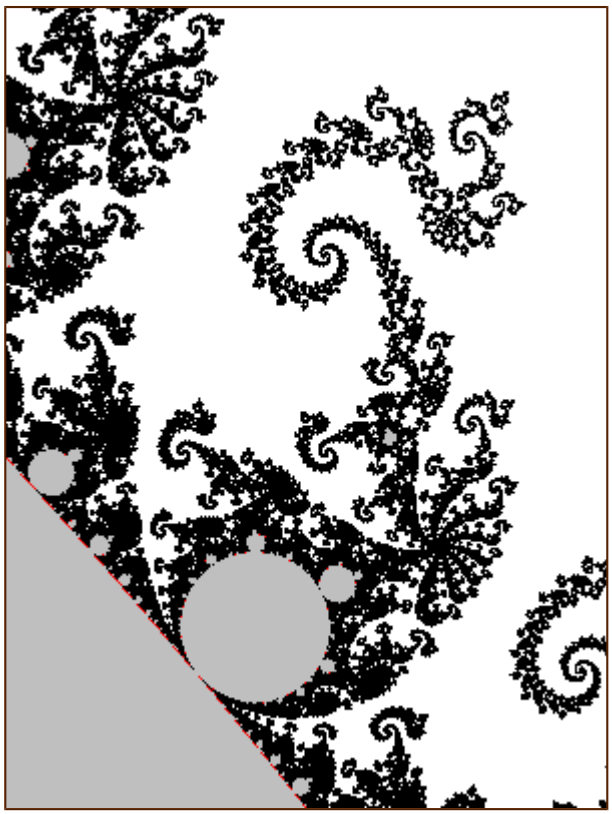


max iter = 100

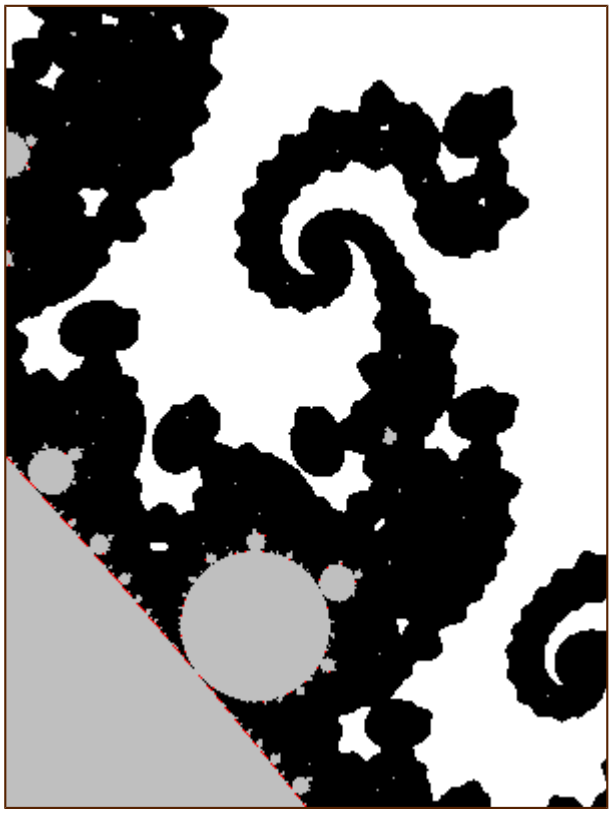


max iter = 100

Note that can choose an oversized value of thickness_factor and the result is not too bad:

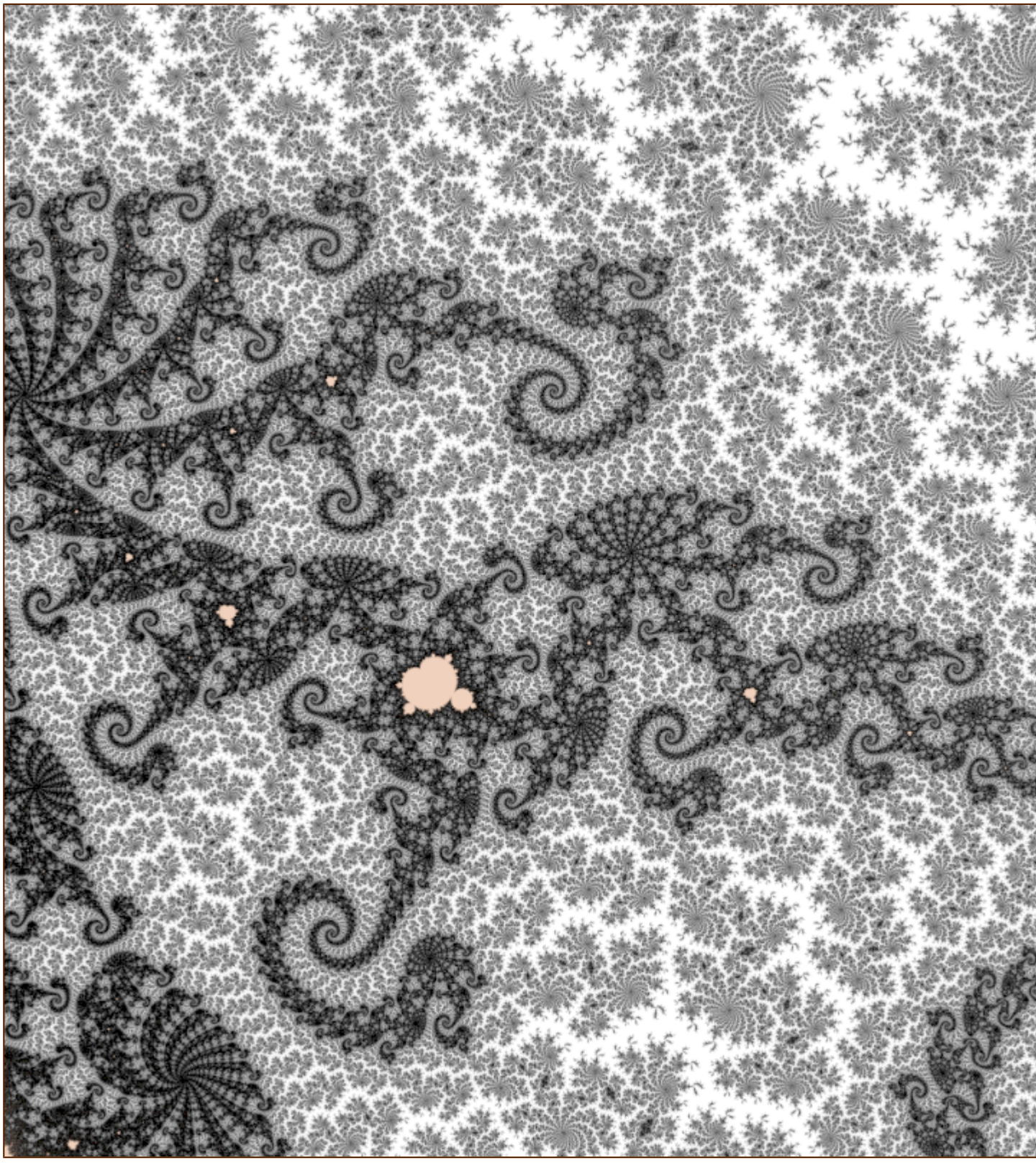


here thickness_factor=0.5



here we took thickness_factor=5.656

To finish, let us show a nice experiment: below we chose a zoom in a place where M is very dense. The picture would be mostly black. We then set the thickness factor to a small vallue (I think it was 0.03) and computed an image of 8000x8000 pixels, that we downscaled to 800x800. We paid extra attention at performing a gamma-correct downscale. The result is quite interesting.



Variation: (partial) antialias effect without oversampling

Recall that we have an estimate *dn*dn for the distance from *z*0z0 to M, where *n*n is the first index for which |*zn*|>*R*|zn| >R. In the previous algorithm we compared this estimate to a fixed size s and set the pixel black if *dn*<*s*dn<s or white otherwise. The idea here is to use the quotient *dn*/*s*dn/s as an interpolation factor from black to white.

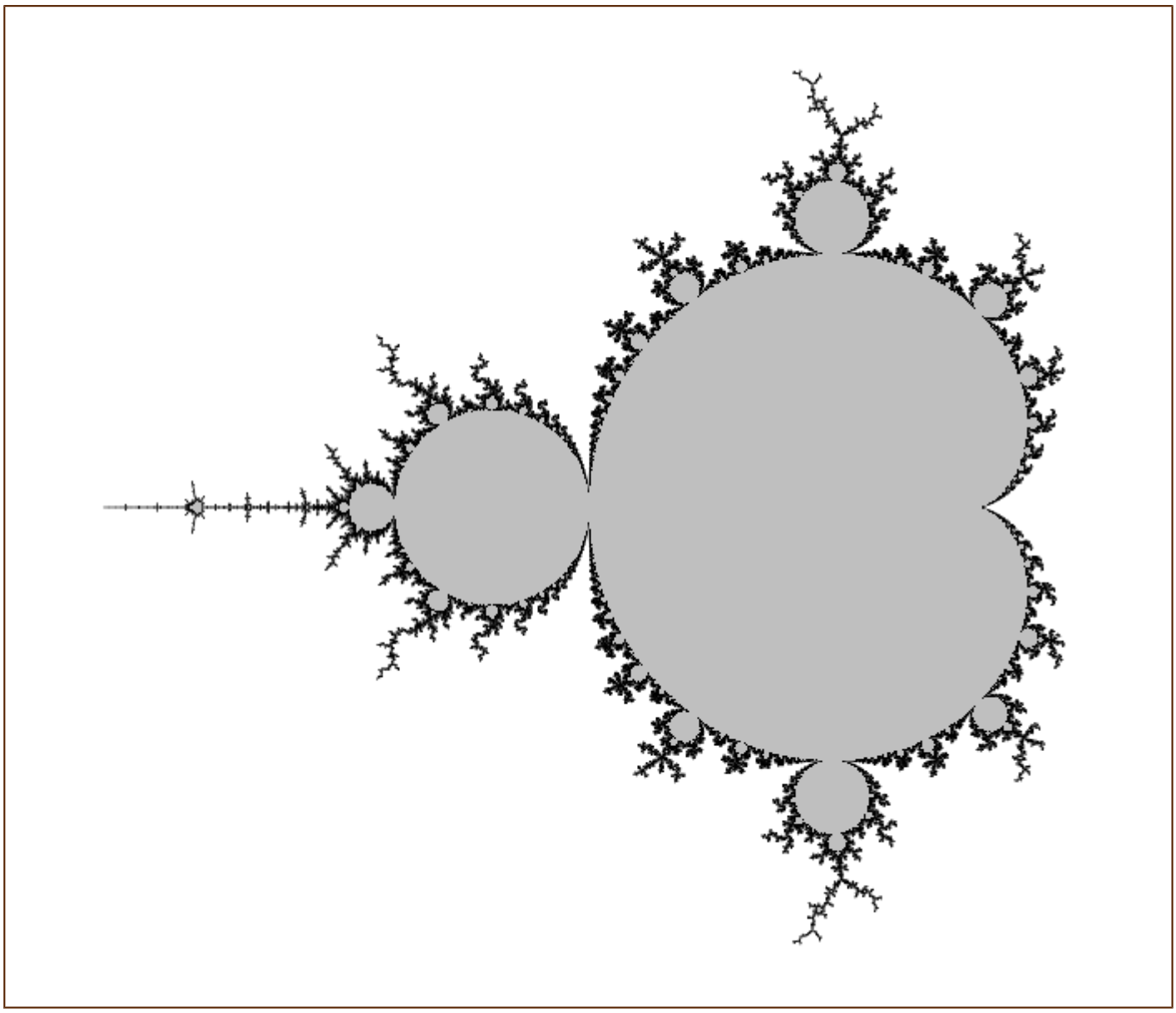
The algorithm should look like this:

```
thickness_factor = 1.414

for p in allpixels:
  c = p.affix
```

```
z = c
der_c = 1+0j
reason = NOT_ENOUGH_ITERATES

for n in range(0,N):
  if squared_modulus(z) > R*R:
    reason = OUTSIDE
    break
  new_z = z*z+c
  new_der_c = der_c*2*z + 1
  z = new_z
  der_c = new_der_c

if reason == NOT_ENOUGH_ITERATES:
  p.color = not_enough_iterates_color
else:
  r = abs(z) # modulus of the complex number
  d = r*2.*log(r) / modulus(der_c)
  t = d / (thickness_factor*pixel_size)
  if(t>1) t=1;
    p.color = linear_interpolation(boundary_color,outside_color,t)
```

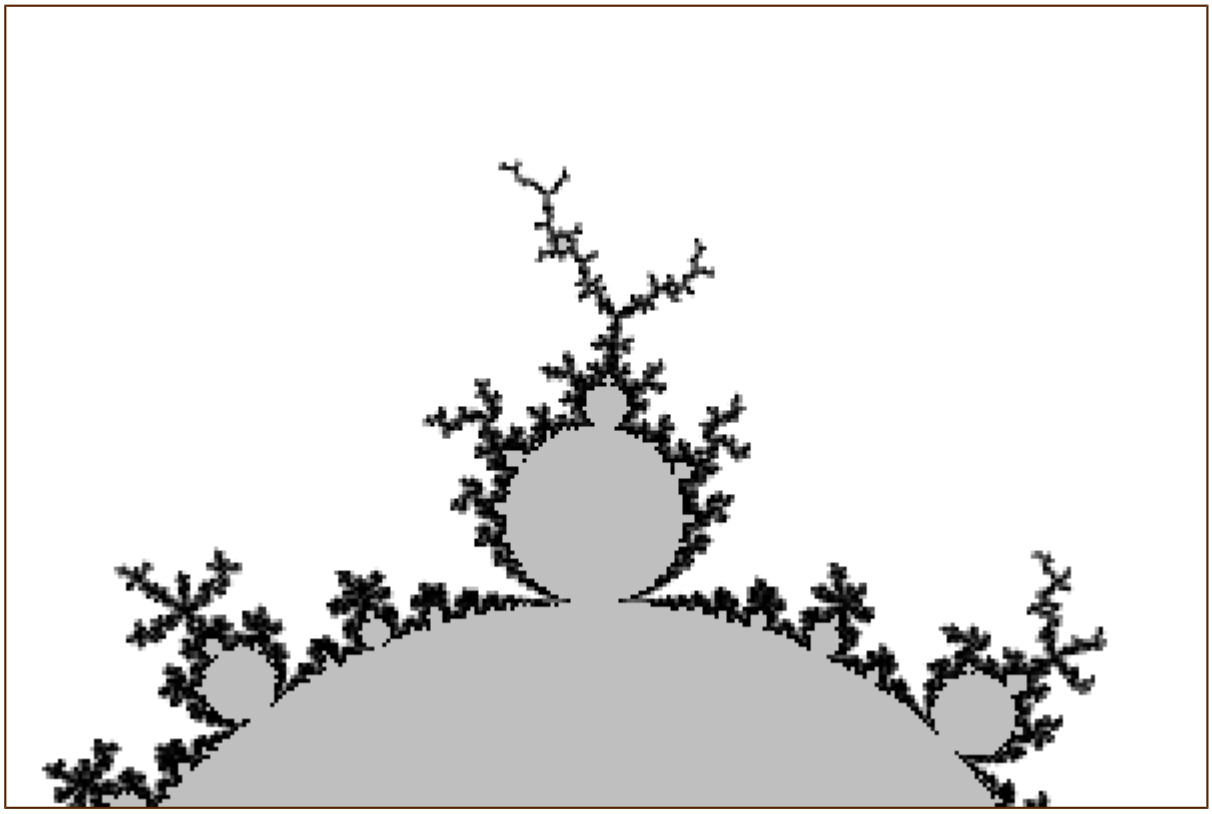
Of course you can add an interior detection method as explained earlier to speed-up things.

Result:

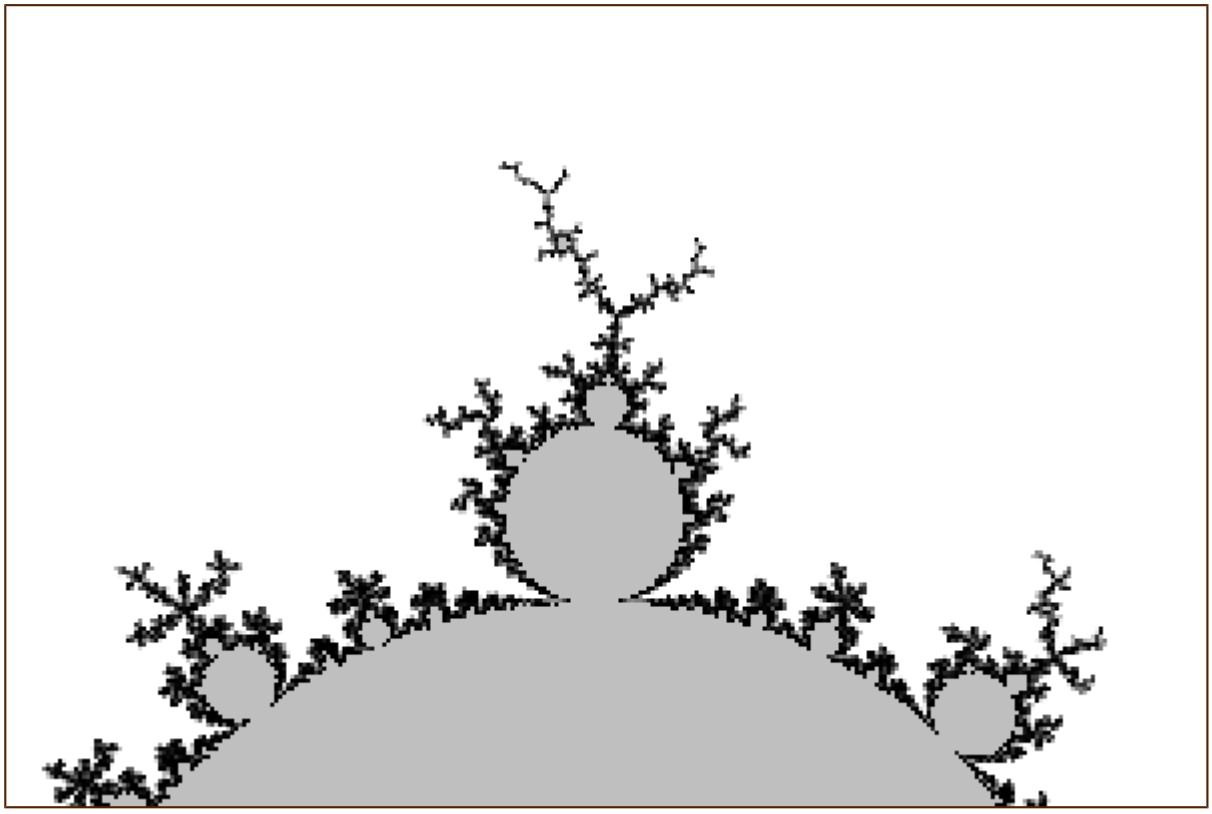


R=1000, thickness factor = 1.414

And a zoom:



thickness factor = 1.414



thickness factor = 1

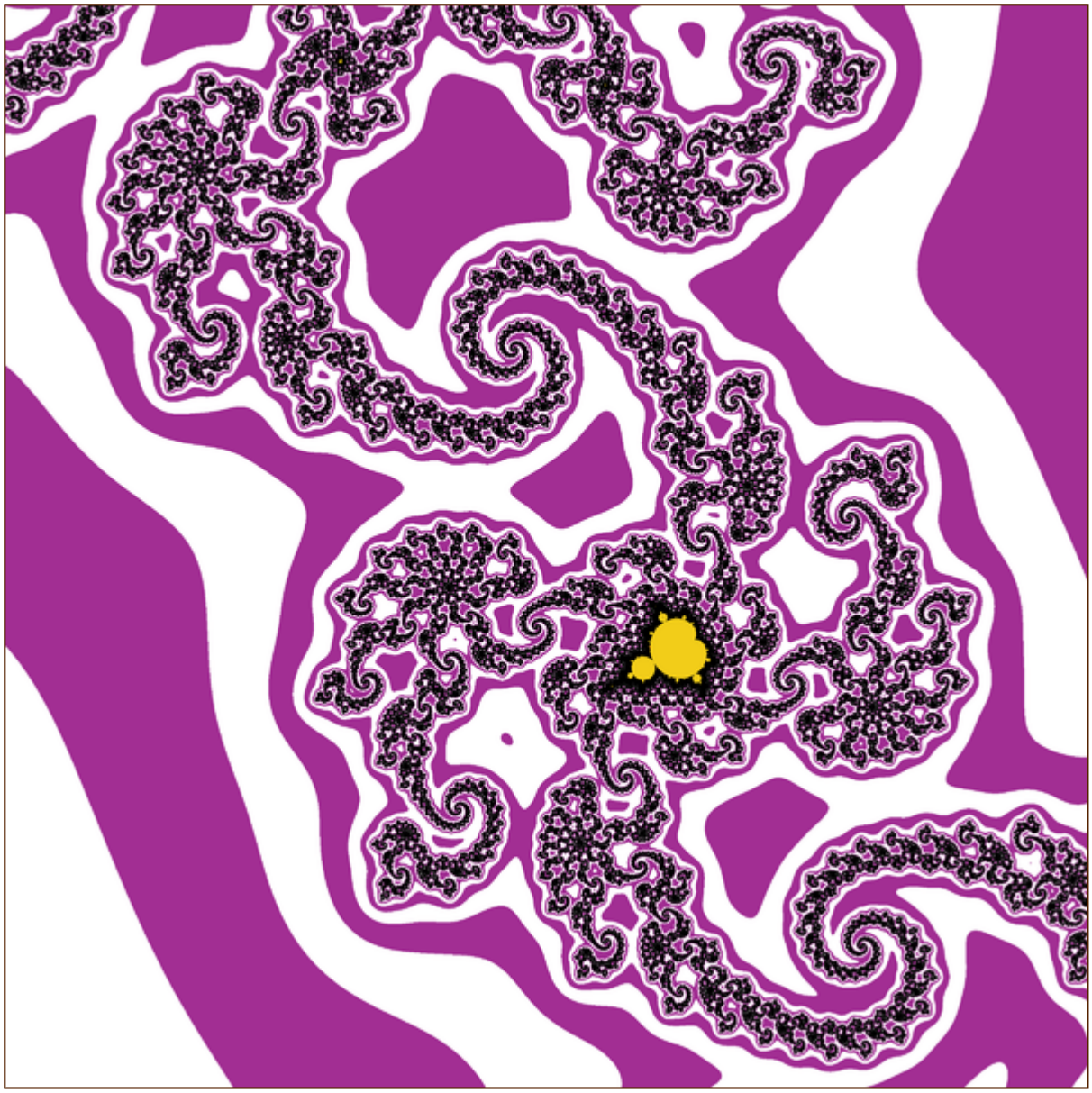
Notice how the algorithm smooths things on the outside but not on the inside.

This is a very simple way to get (partial) antialias without oversampling.

There are possible refinements where one would use a different function of *dn*/*s*dn/s, perhaps beginning with an interval where it is null, perhaps a smooth function, etc... I have not tried them.

Variation: using the distance estimator to color the outside

We can color the outside as an arbitrary function of our estimate d_n of d: here I show a one example without much explanations, there are lots of possible choices.



color = black if close to 0, otherwise alternating bands of white and violet, according to the parity of the integer part of log(d)/constant for some constant

Henriksen's

I do not know who invented this method, but it was taught to me by Christian Henriksen, maybe he invented it?

The method is extremely versatile in that it adapts to most other situations (bifurcation loci of other families of maps, Julia sets of other maps), whereas it is not easy to adapt Milnor's method. In fact, I use Henriksen's method in nearly all my drawing programs in holomorphic dynamics.

Imagine we try to follow the image of a pixel centered on a point zz. If the pixel is small and we are not too close to a critical point of the map, then its image is approximately a square, because the map is conformal. Its size has been multiplied by the modulus of f'(z), and it has been rotated by arg(f'(z)). It will be simpler to approximate the pixel by a disk, so that we do not have to worry about the rotation. Now the idea is to do as if the image of the disk of center z and radius ε was systematically the disk with center f(z) and radius ε×|f'(z)|. This is a bit daring, as it gets more and more wrong when the size gets from a pixel's to a macroscopic size. Also, near critical points, this will be quite bold.

The second idea is to use the fact that ∂M is contained in the closure of the set of values of c for which the critical point is periodic. In fact the closure of those values is the union of ∂M and of one point in each hyperbolic component (this point is called the center of the hyperbolic component) so the algorithm will detect a little bit more than ∂M.

```
thickness_factor = 0.25 # change this if you want to tune the boundary thickness

for p in allpixels:
  c = p.affix
  z = c
  dc = 1+0j
  der = dc
  reason = NOT_ENOUGH_ITERATES

  for n in range(0,N):
    if squared_modulus(z) < squared_modulus(pixel_size * thickness_factor * der):
      reason = BOUNDARY
      break
    if squared_modulus(z) > R*R:
      reason = OUTSIDE
      break
    new_z = z*z+c
    new_der = der*2*z + dc # the reason why we replaced 1 by dc is explained in
the note
    z = new_z
    der = new_der

  if reason == NOT_ENOUGH_ITERATES:
    p.color = not_enough_iterates_color
  elif reason == BOUNDARY:
    p.color = boundary_color
  else: # in this case reason = OUTSIDE
    p.color = outside_color
```

Here, R shall not be too big (R=10 is a good value). Also the thickness factor shall be smaller than in the previous algorithms. It would seem normal to take half of the pixel size as the radius of the ball, or slightly more, but the images turn out to be bad for values >0.25.

Note: A trick to accelerate the computation is to include the factor "pixel_size * thickness_factor" into the definition of dc

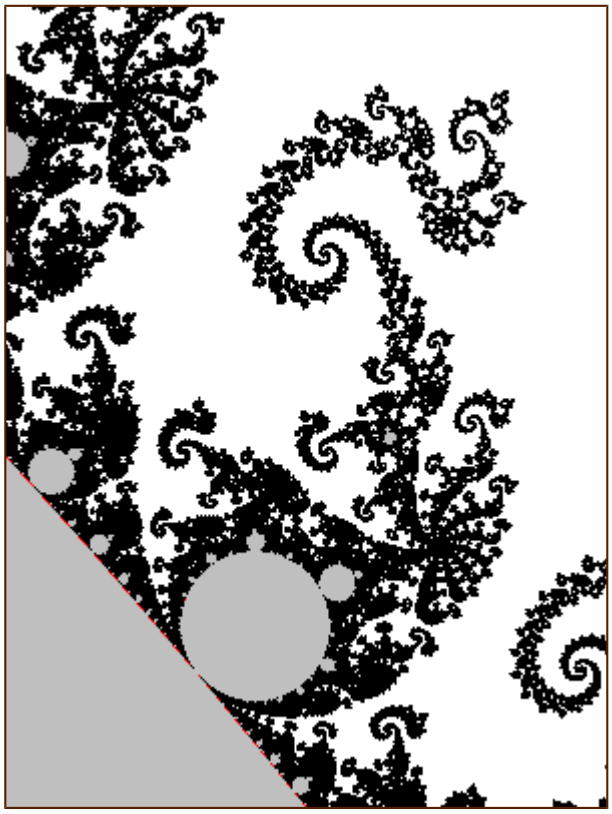
```
  dc = factor pixel_size * thickness_factor * 1+0j
```

so that the first 'if' becomes

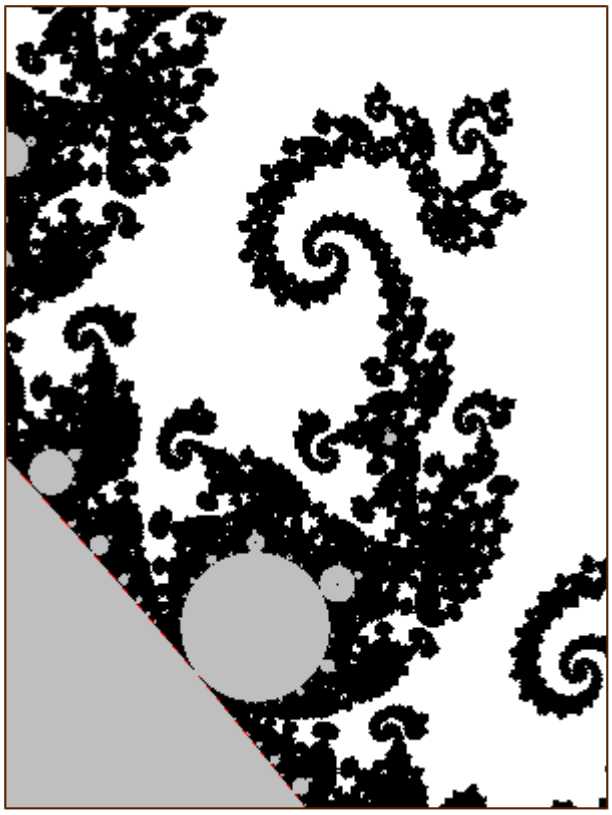
```
    if squared_modulus(z) < squared_modulus(der):
```

i.e. we save a multiplication. This is what I usually do.

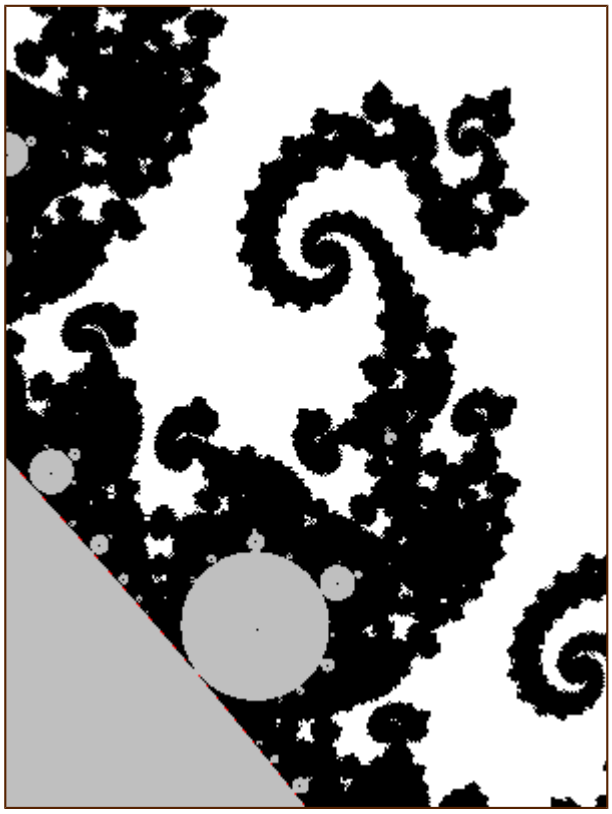
Below we implemented the algorithm above plus interior detection as described above (this mostly accelerate the computation and barely changes the image):



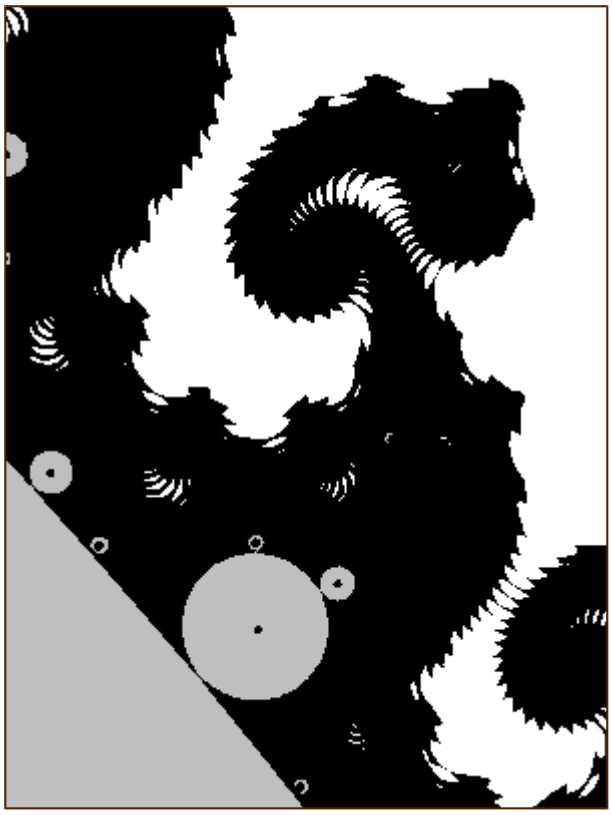
thickness factor = 0.1



thickness factor = 0.25



thickness factor = 0.5



thickness factor = 2

See how the components center are detected and painted black. See also how the hyperbolic component boundaries are nicely drawn. It seems to be due to the fact that only the biggest bulbs attached to them are left gray. Perhaps because the black dots a the center of the small ones cover them completely?

Variant

There is a simple way to avoid detecting the centers of hyperbolic components in our particular case where $Pc(z)=z2+c$Pc(z)=z2+c. Replace the test $Pnc(0)=0$Pcn(0)=0 by $Pnc(0)=-c$Pcn(0)=−c. This corresponding set is now contained in ∂M and is still dense.

In the algorithm above, replace the test

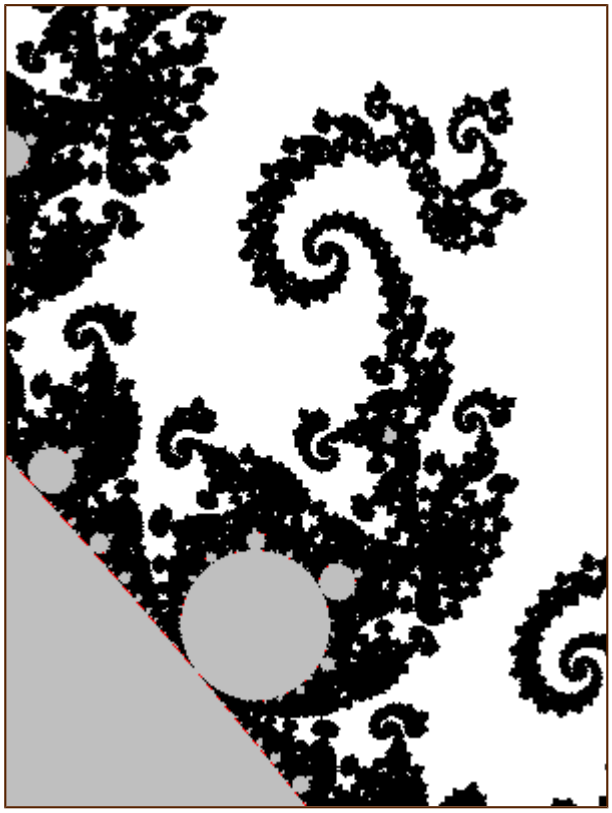
```
if squared_modulus(z) < squared_modulus(der):
```

by

```
if squared_modulus(z+c) < squared_modulus(der+dc):
```

Note that not only we change z to z+c but also der to der+dc. Indeed, the derivative with respect to c of z_n + c is 1 plus the derivative of z_n w.r.t c. Making the mistake here turns out not to be catastrophic, though I do not know why.

Results:



thickness factor = 0.25

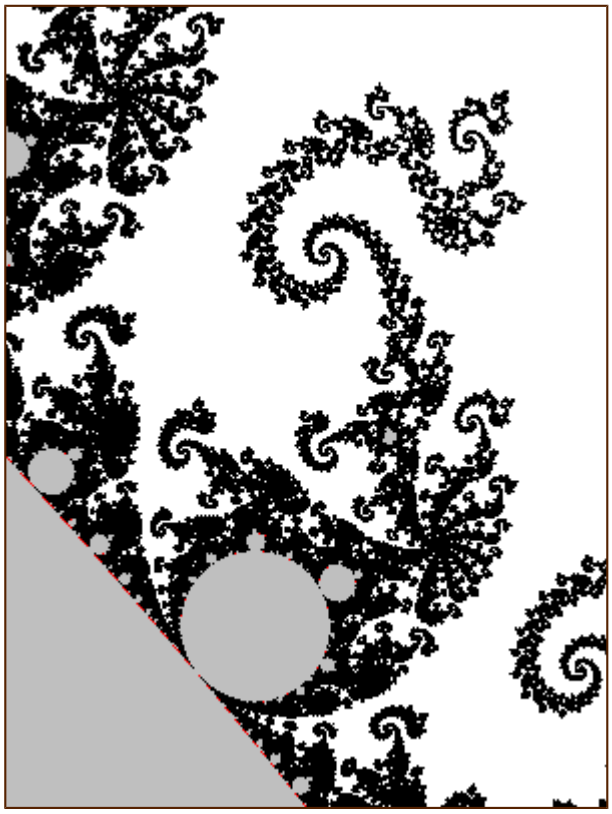
Another variant

We will use the test z+c=0 but using the spherical metric instead of the euclidean one.

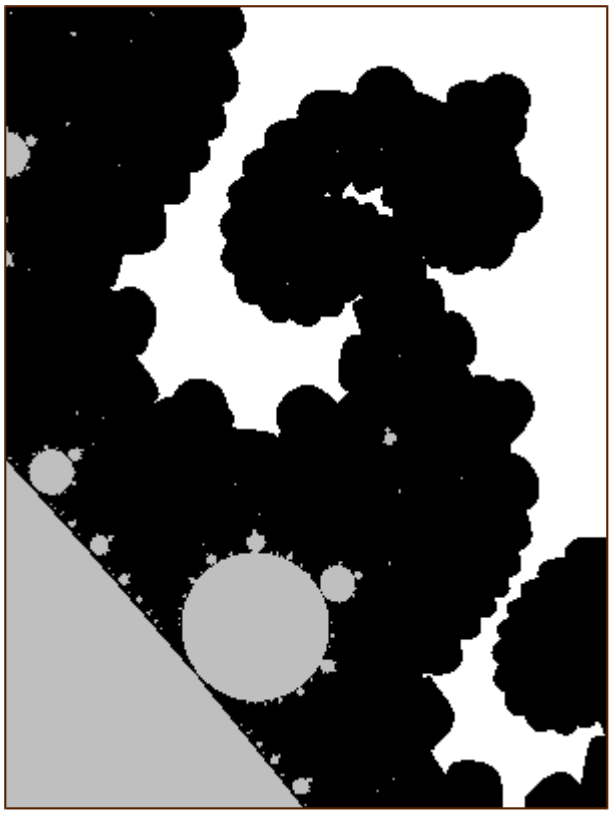
```
rsq = squared_modulus(z+c)
if rsq*(1+rsq*0.25) < squared_modulus(der+dc):
```

One may take another constant than 0.25 (this amounts to changing the equator of the sphere).

Result (again we used interior detection and again, this barely changes the image):



thickness factor = 0.5



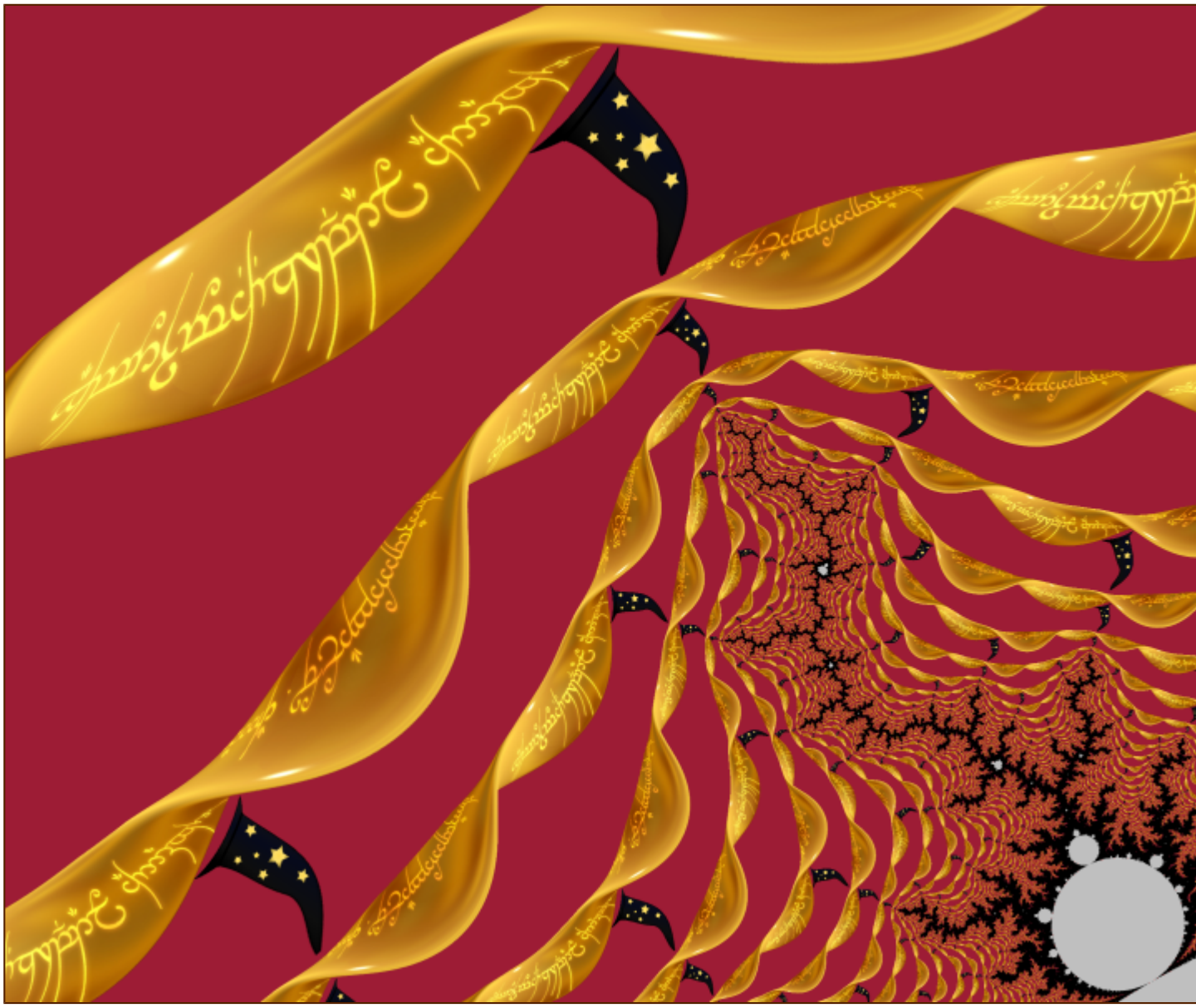
thickness factor = 20

Fancy coloring of the outside

Images

Here we look at the first value of z whose modulus goes beyond R and use its value as a coordinate in a square base image to retrieve a pixel whose color we use.

Below, this was mixed with boundary detection and with interior detection.



Above, we took R=4.8 and used the following base image, a collage of public domain images:

See also Paul Nylander's page on fractals, in particular the item named *Mandelbrot set tesselation*.

Normal map effect

Normal maps are components of 2D texture used in 3D rendering. Fine scale relief on flat surfaces can be visible with proper lighting. They can be approximated by defining (artificially) a normal vector for each point of the texture, usually by a 2D array of values as for a color texture. Given the normal and the direction of light (and possibly the observer direction) one deduces a first approximation of the shading effect, which is already pretty good.

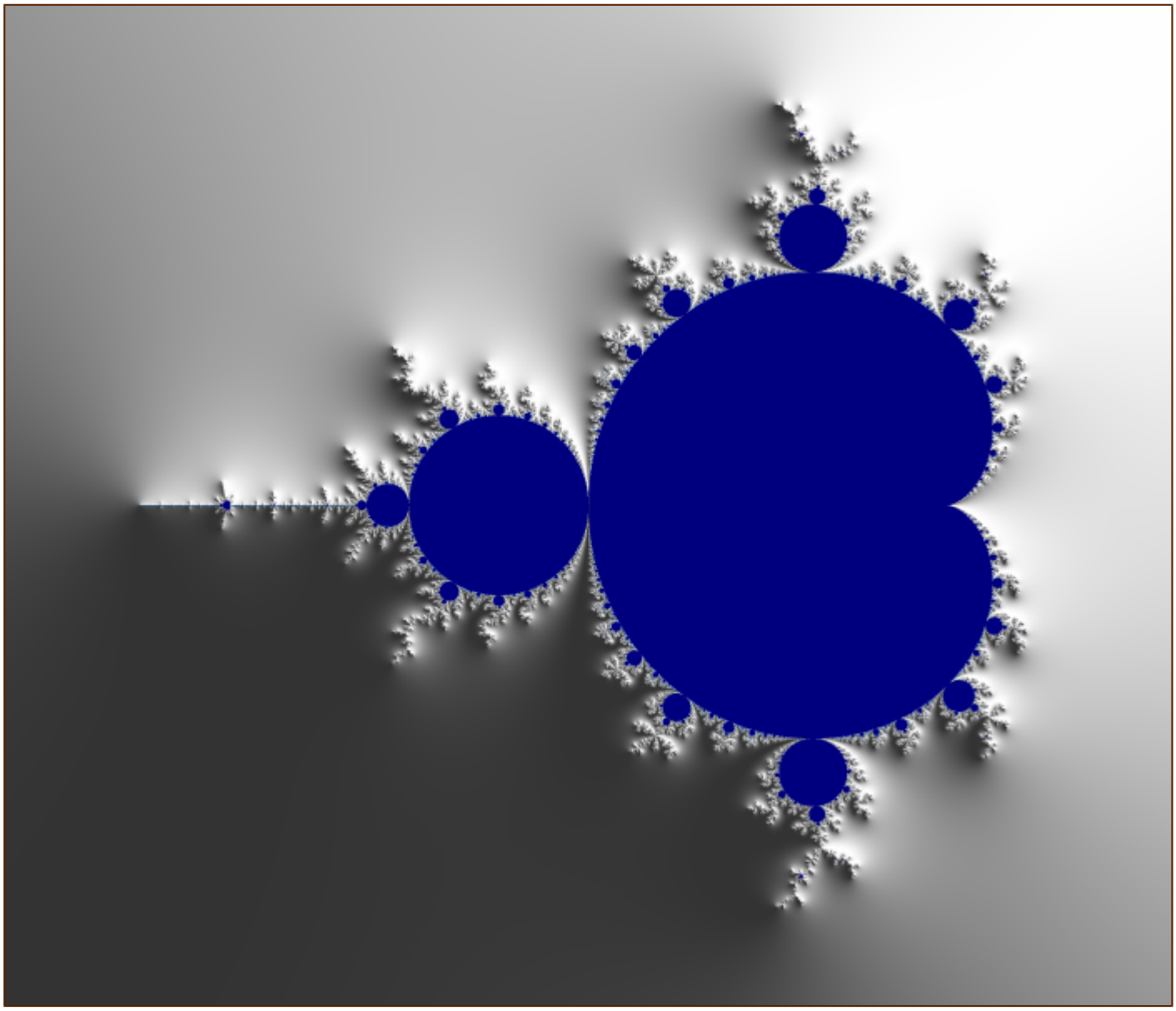
Here we define a normal (only) for a point on the outside of M as the vector of coordinates (x,y,1)/&sqrt;2 where (x,y) is the normal to the potential line through the point. Since after reaching $|zn|>R$|zn|>R (for R big enough) the potential is approximated by $2-nlog|zn|$2−nlog|zn|, one just has to pull-back the radial direction by the derivative of $c\mapsto zn$c↦zn, i.e. do z/der in the notation of the scripts, to get a vector (x,y) normal to the equipotential. We then normalize this vector so it has modulus one and use the simplest model for the shading: Lambert, which consists in using the dot product of (x,y,1) with a constant vector indicating the direction of the light.

```
h2 = 1.5  # height factor of the incoming light
angle = 45  # incoming direction of light
v = exp(1j*angle*2*pi/360)  # unit 2D vector in this direction
# incoming light 3D vector = (v.re,v.im,h2)

R = 100  # do not take R too small

for p in allpixels:
  c = p.affix
  z = c
  dc = 1+0j
  der = dc
  reason = NOT_ENOUGH_ITERATES

  for n in range(0,N):
    if squared_modulus(z) > R*R:
      reason = OUTSIDE
      break
    new_z = z*z+c
    new_der = der*2*z + dc
    z = new_z
    der = new_der

  if reason == NOT_ENOUGH_ITERATES:
    p.color = not_enough_iterates_color
  else:  # in this case reason = OUTSIDE
    u = z/der
    u = u/abs(u)  # normal vector: (u.re,u.im,1)
    t = u.real*v.real + u.imag*v.imag + h2  # dot product with the incoming light
    t = t/(1+h2)  # rescale so that t does not get bigger than 1
    if t<0: t=0
    p.color = linear_interpolation(black,white,t)
```

If h2<1 some parts can be in total darkness, the dot product being negative, which is why we have the line "if t<0: t=0". One can refine this by setting a back light or an ambiant light effect. Also, there are more sophisticated lighting models, like the Oren Nayar for rough surfaces.

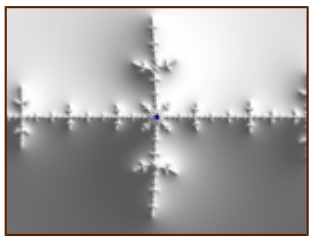
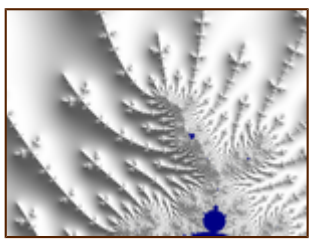
It is important to note that our normal map is not realistic in the sense that there can be no height field whose normals are in these directions (by lack of integrability). We could try to ensure that but I suspect that this would result in a too flat image near M.

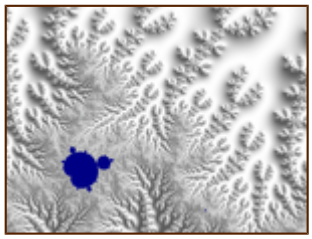
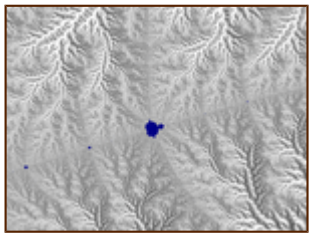
Results: first the whole thing.



downscaled by 4

A small gallery:

- 
- 

- 
- 

Variation

We can also use Milnor's distance estimator in place of the potential. The computation is slightly more complicated.

We need to compute the direction of level sets of the function d, where d is the distance estimator. The function d itself makes use of the derivative of $z_n$zn w.r.t c and we will need to differentiate once more. Recall:

$$dn=|z_n|\log|z_n||dz_n/dc|.\text{dn}=|\text{zn}|\log|\text{zn}||\text{dzn/dc}|.$$

This is *not* a holomorphic function. To compute its total differential dd_nddn, we need to compute the total differentials $d|z_n|$d|zn| and $d|dz_n/dc|$d|dzn/dc|: (i'm omitting the index n below)

$$dd=1+\log|z||dz/dc|d|z|-|z|\log|z||dz/dc|2d|dz/dc|\text{dd}=1+\log|\text{z}||\text{dz/dc}|\text{d}|\text{z}|-|\text{z}|\log|\text{z}||\text{dz/dc}|2\text{d}|\text{dz/dc}|$$

Now for any complex valued variable or function aa, not necessarily holomorphic, we have $d|a|=(a\cdot da)/|a|$d|a|=(a·da)/|a| where the dot product $a\cdot b$a·b can be computed either using $\text{Re}(a)\text{Re}(b)+\text{Im}(a)\text{Im}(b)$Re(a)Re(b)+Im(a)Im(b) or $\text{Re}(ab^-)\text{Re}(\text{ab}^-)$. The second form computes an imaginary part that will not be used so is less useful in a program, however it can be useful while doing algebraic computations (for this respect we also have the form $(ab^-+a^-b)/2$(ab¯+a¯b)/2, which we won't use here).

Last, $d(dz/dc)$d(dz/dc) is a holomorphic differential, equal to $d2z_dc2dc$d2zdc2dc. So along the iteration we will need to compute not only $z_n$zn and dz_n/dcdzn/dc but also $d2z_n/dc2$d2zn/dc2. This can be done as follows:

```
for p in allpixels:
  [...]
  z = c
  der = 1
  der2 = 0
  [...]

  for n in range(0,N):
    [...]
    new_z = z*z+c
    new_der = 2*der*z + 1
    new_der2 = 2*(der2*z+der**2)
    z = new_z
    der = new_der
    der2 = new_der2

    [...]
```

Let us now expand the mathematical computation of the direction vector:

$$dd=1+\log|z||dz/dc|z\cdot dz|z|-|z|\log|z||dz/dc|2dz/dc\cdot d(dz/dc)|dz/dc|\text{dd}=1+\log|\text{z}||\text{dz/dc}|\text{z}\cdot\text{dz}|\text{z}|-|\text{z}|\log|\text{z}||\text{dz/dc}|$$
$$2\text{dz/dc}\cdot\text{d(dz/dc)}|\text{dz/dc}|$$

This is getting complicated and typing mistakes are likely to occur. Let us use $a=dz/dc$a=dz/dc and $b=d2z/dc2$b=d2z/dc2: these are the quantities that we compute above along with z. Since $d(dz/dc)=(d2z/dc2)dc=bdc$d(dz/dc)=(d2z/dc2)dc=bdc we get

$$dd=1+\log|z||a|z\cdot(adc)|z|-|z|\log|z||a|2a\cdot(bdc)|a|$$dd=1+log|z||a|z·(adc)|z|−|z|log|z||a|2a·(bdc)|a|

whence

$$dd=\mathrm{Re}\left(\left(1+\log|z||a|z\bar{a}|z|-|z|\log|z||a|2a\bar{b}|a|\right)dc\overline{}\right)$$dd=Re((1+log|z||a|zā|z|−|z|log|z||a|2ab̄|a|)dc‾)

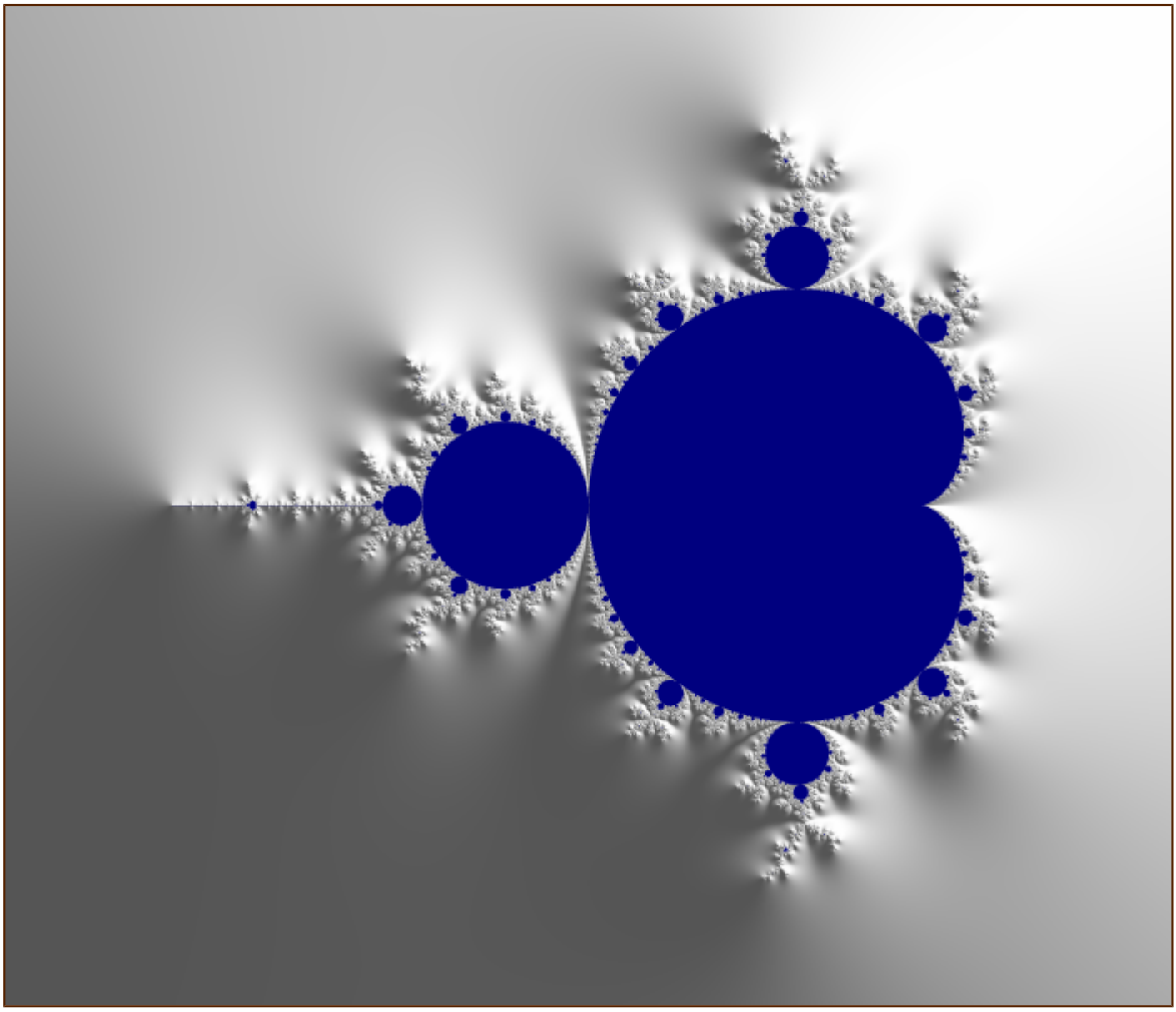
The complex number that $dc\overline{}$dc‾ is multiplied with points in the direction we are looking for, i.e. the direction of the normal to the level set of the function d. We just need the direction so we may well multiply it to get rid of denominators (multiplication is slightly faster than division on CPUs; in fact here it is not critical to optimize the computation since most of the time is likely to be spent in the core loop): so we will use the vector

$$u=\left(|a|2(1+\log|z|)\right)z\bar{a}-\left(|z|2\log|z|\right)a\bar{b}$$.u=(|a|2(1+log|z|))zā−(|z|2log|z|)ab̄.

Once the point z escapes we thus use the computation:

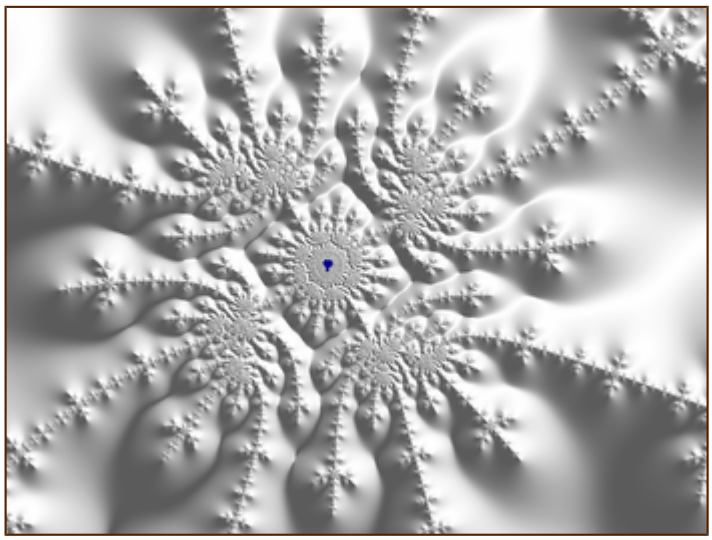
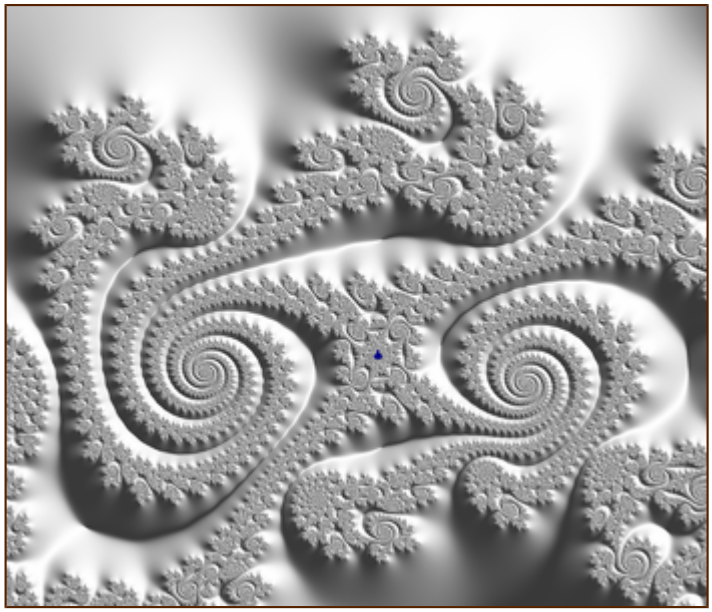
```
[...]
else:  # in this case reason = OUTSIDE
  lo = 0.5*log(squared_modulus(z))
  u = z*der*((1+lo)*conj(der**2)-lo*conj(z*der2))
  u = u/abs(u)
  [...]
```

then proceed as above.



downscaled by 4

Some results:

- 
- 

More variations

Plenty! For instance one can perturb the normal using the argument of *zn*zn, maybe with an averaging method (see below).

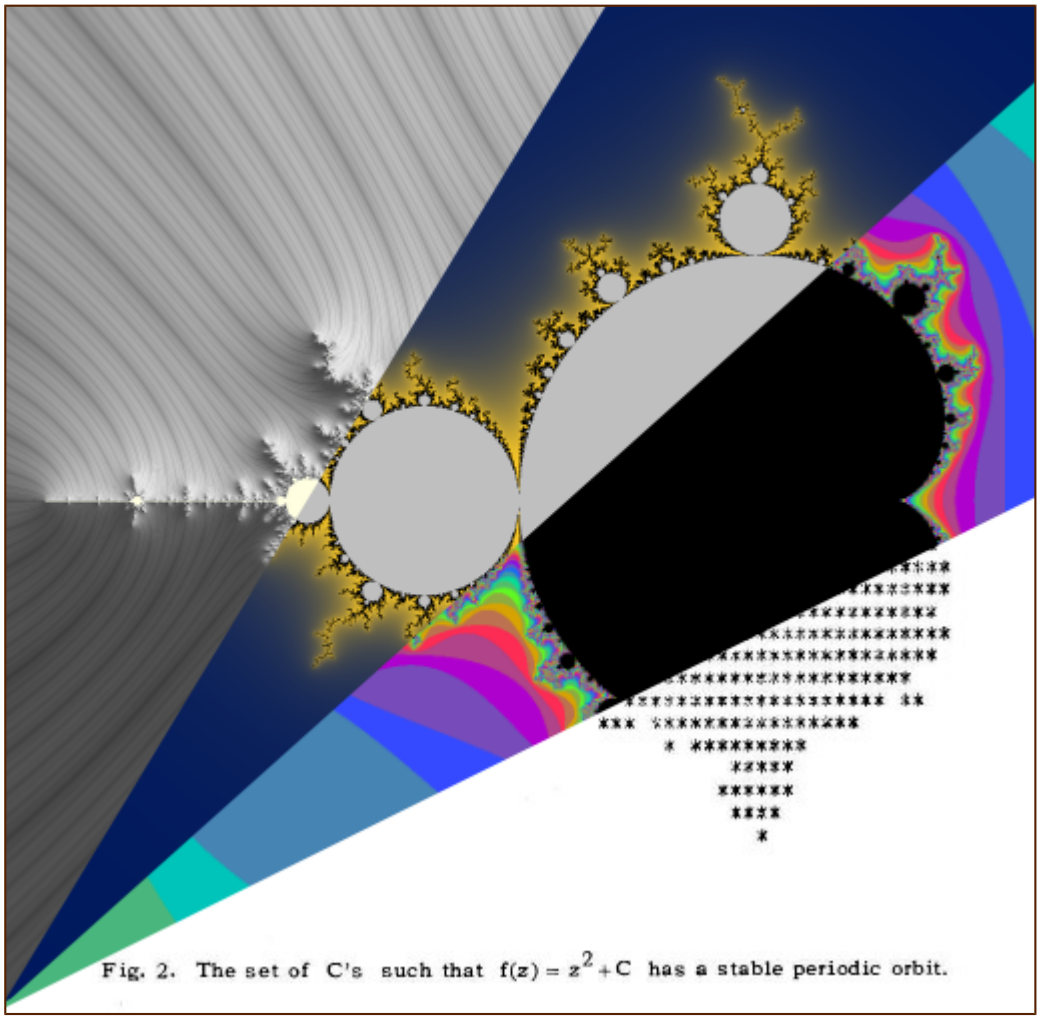
Radial strands

I saw on the Internet a method called the *Triangle Inequality Average*. The method is due to Kerry Mitchell. I prefer the variant called *Stripe Average*, see [3]. It is less pretty but follows the external rays which are of great importance in the theoretical study of M and in holomorphic dynamics.

Mixing it all

As a concluding section, let us mention that one can mix the different techniques, either programmatically in the algorithm, or after producing a certain number of images of the same part, by merging them with an image editor program. Using layers and transparency, one can achieve extremely pretty pictures, as can be found by an Internet search.

Or we can make a patchwork illustrating the evolution of different representation techniques.



Logo of an article

I made the picture above for the following article: http://images.math.cnrs.fr/L-ensemble-de-Mandelbrot.html

References

1. Jump up ↑ Dynamics in one complex variable
2. Jump up ↑ Peitgen and Saupe, The art of fractal programming, 1986
3. Jump up ↑ On fractal coloring techniques, Master's Thesis by Jussi Härkönen, 2007

Navigation menu

Personal tools

- Log in

Namespaces

- Page
- Discussion

Variants

Views

- Read
- View source
- View history

More

Search

Go

Navigation

- Main page
- Recent changes
- Random page
- Help

Tools

- What links here
- Related changes
- Special pages
- Permanent link
- Page information

- About Techniques for computer generated pictures in complex dynamics