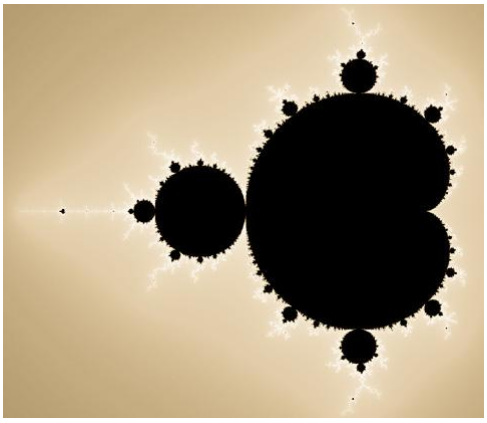
Boldly Going - Mandelbrot Set Area

Welcome to a new article in my *"Boldly Going"* series, this time starring the **Mandelbrot set** and the difficult task of computing an accurate estimation of its area. The task is fraught with difficulties and it's been attacked with really powerful hardware (think 4 GPUs), complex software and extremely long computation times (think 35 days) but all that work has produced only about 8-9 correct digits. Here I'll attempt the feat using just my trusty HP calculators, many orders of magnitude slower and less capable but nevertheless I'll manage to get about 5-6 correct digits in much shorter times.

Introduction

The Mandelbrot set (**M** for short) is the most well-known fractal of all, an amazing mathematical object which mystified everyone since its discovery by B. Mandelbrot ca. 1975 and subsequent popularization in the August 1985 issue of *Scientific American*. There is an incredible amount of readily available literature dealing with all aspects of **M** from the very basic to the most advanced so I'll refer the reader to it and won't discuss them here.



M has a fractal boundary which encloses a finite area whose precise value is still an open question, and an estimation of it is what this article is all about. To wit, there are several ways to try and estimate the area, including[1]:

- the ***Monte Carlo*** approach, where a large number of random points are generated within some enclosing box, and a tally is kept of how many belong to **M**, which is then used to compute the estimation.
- the ***pixel-counting*** approach, where finer and finer grids are averaged to tally the number of grid points belonging to **M**.
- the ***theoretical*** approach, where a large number of terms of an exact formula converging (extremely slowly) to the area of **M** are evaluated and added up to get an estimate.

The ***Monte Carlo*** approach has some advantages (such as not being prone to potential aliasing problems as may happen with equally-spaced grids) and disadvantages, the main one being that as is typical of standard *Monte Carlo* approaches, to get one more correct digit (i.e., increasing the resolution 10x) the number of generated pixels would need to be increased 100x, which would result in approximately 100x the running time. It also requires a very good, non-biased random number generator with a large cycle (at least several billions long).

The ***pixel-counting*** approach has been widely used. For example, back in 2012 R. Munafo launched an 8-day run to calculate almost 17 trillion pixels (at 2.4 million px/sec) to get an estimated area of ***1.506591856*** with an estimated error of *0.0000000256*.

Later, T. Förstemann used some powerful hardware (*Intel Core i7 2600K* CPU, 2x GPU *Radeon HD 5970* for a total of 4 GPUs with 1600 stream processors each, 350W under load) and software (*Mathematica 8.0.4.0* under *Windows 7*, *ATI* driver *Catalyst 11.2* with *AMD Stream SDK 2.3* and installation of a C-compiler [*Visual Studio 2011*] for *Mathematica*) running for 35 days straight with a grid size of *2,097,152* for a total of *87,960,930,222,520* calculated pixels (at more than 29 million px/sec and depths starting at *8,589,934,592* iterations) to get an estimated area/error of ***1.5065918849*** and *0.0000000028*, ten times better than Munafo's.

[1] Other methods include the *μ-atom method*, used by J. Hill to get a lower bound which is close to the pixel counting methods. He included the area of all components up to period 16 (*main cardioid* is *P1*, *main disk* is *P2*), and all of period 16 but one, and got an area of ***1.506303622***, which differs from Förstemann's by ~ *0.0002883 (0.019%)*.

Finally, the ***theoretical*** approach uses *Laurent Series*, in particular a specific one introduced by Ewing and Schober, which allows computing the area of **M** by evaluating an infinite series of the form:

$$\boldsymbol{M}_{area} = \pi \left(1 - \sum_{n=0}^{\infty} n.{b_n}^2\right)$$

where b_n are the coefficients of the Laurent series, the first ones being $b_0 = -1/2$, $b_1 = 1/8$, $b_2 = -1/4$, $b_3 = 15/128$, $b_4 = 0$, $b_5 = -47/1024$, etc. For a finite number of terms this formula always gives an upper bound of the area but despite its mathematical elegance it is absolutely unsuitable to compute the area as it converges incredibly *slowly*, with an estimated $6.4.10^{11}$ terms needed to get just *one* correct digit and more than 10^{118} terms to get *two* !

Matter of fact, Ewing *et al* used *500,000 terms* ($b_{500000} \sim 5.5221313 \cdot 10^{-8}$) in 1990 to get an estimated area of ***1.72*** and later in 2014 Bittner *et al* used *5,000,000* terms (whose b_n coefficients took 3 months to compute, $b_{5000000} \sim 8.0532 \cdot 10^{-11}$) and got an estimation of ***1.68288***.

To complicate the matter even further, this theoretical approach seems to converge to a value between *1.60* and *1.70* while the empirical approaches (*Monte Carlo* and pixel counting) give estimates around ***1.50659***. This might be due to the fact that the *boundary* of **M** has *Hausdorff* dimension *2* and thus *might* have positive (i.e., non-zero) area, which would account for the discrepancy as none of the empirical approaches can ever generate and calculate points or pixels exactly belonging to **M**'s boundary, so their potential contribution to the area would never be included in the computation. As of 2020, this is still in the realm of speculation but nevertheless it seems quite plausible[1].

Boldly going ...

As stated in the *Introduction* above, the purpose of this article is to use nothing but my trusty HP calculators (whether in physical or virtual form) to try and compute an estimation as accurate as possible (say 5-6 correct digits) for **M**'s area in reasonable times: less than half an hour for a virtual calc, a day or two at most for a physical one), which is no mean feat.

In view of the above described hardware, software and computation time requirements, it's clear that accomplishing my goal will require a good algorithm and pretty optimized code. As this is an informal Article, not a formal research paper, I'll adopt a *Machiavellian* approach *("The Ends Justify the Means")* and I'll mix sound mathematical optimizations with more informal heuristics as required.

To begin with, I'll use a ***Monte Carlo*** approach, generating a suitably large number *N* of random points within a rectangular box which completely encloses **M**, and counting how many actually belong to **M**. The sought-for area will then be proportional to the count. To make the task manageable I'll use the following optimizations:

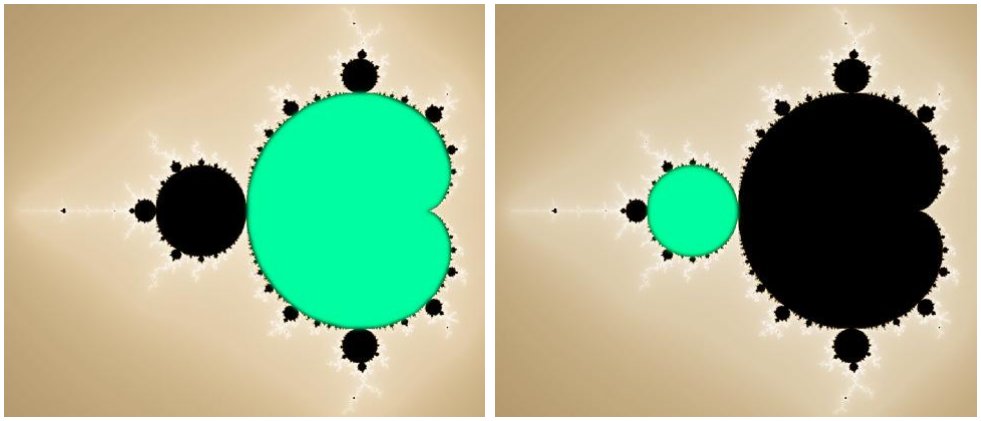
- Each point *(x,y)* will be generated as a random complex number *z* within a rectangular box enclosing **M.** Actually, the leftmost extreme of **M** is at $x = -2$, the righmost extreme is at $x = 0.471185334933396+$, the topmost extreme is at $y = 1.122757063632597+$ and the downmost extreme is at $y = -1.122757063632597+$.

- As **M** is *symmetric*, I only need to compute the area of the top half and the total area of **M** will then be twice this value. This means that I can use a smaller rectangular box with *x* ranging from *-2* to *0.5* and with *y* ranging from *0* to *1.2* and I'll generate all random complex points *z* within that box.

- Each randomly generated complex *z* has to be tested for inclusion in **M**, which is done via the usual *escape time* algorithm: start with $z_0 = (0,0)$ and $c = z$, then iteratively compute $z_{n+1} = z_n^2 + c$ until either the absolute value of $z_n \geq 2$, in which case *z* escapes to infinity and so definitely does *not* belong to **M**, or else a max. number of iterations is reached and *z* is considered to belong to **M** and the count is increased by 1.

[1] D. Allingham (see *References*) wrote: *"B. Mandelbrot himself conjetures that the boundary of the set may have Hausdorff dimension 2, which would imply that it actually contributes to the area."*

- As computing whether every *z* belongs to **M** is a very time-consuming iterative process (which will reach the maximum number of iterations if *z* actually belongs to **M**) we can try and avoid it altogether for those *z* which we can easily ascertain in advance as belonging to **M** without performing any iterations. That's the case for those *z* either in the *main cardioid (below left)* or in the largest circular bud *(main disk, below right)*:



- The main cardioid's area is $3\pi/8 = 1.178097+$ (about *78.20%* of the total area), while the main disk has an area of $\pi/16 = 0.196350+$, (another *13.03%)* and their combined total is $7\pi/16 = 1.374447+$, which already accounts for *91.23%* of the total area of **M** so we need to compute just the remaining *8.77%*, thus the expensive iterative process will be executed in full less than *9%* of the time, a considerable savings.

- To wit, if we can *quickly* check whether a given *z* belongs or not to the main cardioid or the main disk we'll save lots of running time and as it happens, indeed we actually *can*, using just a few steps for the *RPN* version or just 2 lines of code for the *BASIC* version.

- As for those points not belonging to either the main cardioid or the main disk, checking whether they belong to some other minor disks or cardioids quickly becomes more expensive and complicated than performing the *K* iterations, which will proceed faster if *K* is relatively small, say *256* iterations max.

  However, this will adversely affect the accuracy because there will be points which do not escape to infinity in *256* iterations but would if performing *512* iterations, say, and the same would happen with a bigger *K*, there will always be points (i.e.: those sufficiently close to the boundary) which will require more iterations than any limit we might specify in advance and so those points would be miscounted as belonging to **M** while actually they don't. Nevertheless, there will be fewer of them as *K* grows bigger, which will help increase the accuracy but negatively impact the running time.

- I'll attempt to alleviate this dilemma by calculating a large number *N* of random points but using a relatively low maximum number of iterations, say $K = 256$, which will speed the computation as desired. To increase the accuracy, I'll apply afterwards a *correction factor* to the resulting area, which will be heuristically computed like this: we'll choose a suitably smaller number of random points $N_2 << N$ and we'll obtain the count of the points belonging to **M** using first $K = 256$, then $K = 1024$ iterations. The resulting correction factor would then be:

$$f_{corr} = count_{1024} / count_{256}$$

  Simple as it is, this non-rigorous, heuristic approach works quite nicely and will allow us to use a relatively low number of max. iterations without actually compromising the obtained accuracy too much.

- In short, my algorithm will rely on: *(a)* rigorous math (statistically-sound *Monte Carlo* method, tight box, symmetry, main cardioid and disk detection, etc.), *(b)* nonrigorous heuristics (the *correction factor*) and last but not least *(c)* a little *luck*. When dealing with random numbers you always need a little luck, as the sequence *7,7,7, ...* has the same probability as any other more random-looking sequence. In practice this means that the results might be *worse* than average or *better* than average and the latter case is the lucky part.

Program Listing for the HP-71B

308 bytes

```
1  DESTROY ALL @ INPUT "#Points,Iters,Every=";N,K,H @ COMPLEX C,Z @ RANDOMIZE 1
2  H=H+(N+1)*NOT H @ L=1.13 @ M=0 @ FOR I=1 TO N @ C=(RND*2.5-2,RND*L)
3  Z=SGN(C) @ IF 4*ABS(C)<ABS(Z*(2-Z)) THEN 7
4  IF ABS(C+1)<.25 THEN 7
5  Z=0 @ FOR J=1 TO K @ Z=Z*Z+C @ IF ABS(Z)>=2 THEN 8
6  NEXT J
7  M=M+1
8  IF NOT MOD(I,H) THEN DISP USING "2X,2(X,7D),3X,Z.6D";I,M,5*L*M/I
9  NEXT I @ A=5*L*M/N @ DISP USING "/,2X,K,' M-points. Area: ',Z.6D";M,A
```

Program details

Line 1: main entry point: initialization and prompting input from the user.
Line 2: start of the main loop and generation of a random point within the box.
Line 3: checking whether the point belongs to the main cardioid (thus, to **M**).
Line 4: checking whether the point belongs to the main disk (thus, to **M**).
Line 5: checking whether the point belongs elsewhere in **M** (iterations).
Line 6: end of iterations loop.
Line 7: the point does indeed belong to **M**, increment the count.
Line 8: if specified, output intermediate results.
Line 9: end of main loop and output of the final result.

Usage Instructions

The program accepts the number *N* of points to generate, the maximum number of iterations *K*, and whether you want to display intermediate results every *P* points or just the final estimation for the area.

The program doesn't automatically compute/apply any *correction factor*, that's left at the discretion of the user to decide whether and how to compute it since there's no optimal approach valid for all *N* and *K*, there's plenty of leeway. Of course, the program will greatly assist in computing it, as we'll see in the main run below.

To compute an estimation of the area of **M** proceed as follows:

RUN ***#Points,Iters,Every=*** *num.points, max. iterations*[1]*, output results every P points*[2] **END LINE**

→ *Point* $_P$ *Count* $_P$ *Area* $_P$ *{ the intermediate #points generated, count, and estimated area }*
... *{ ditto }*

→ *Count* $_N$ ***#M-points, Area:*** *Area* $_N$ *{ the final count and estimation of the area }*

[1] The number of iterations doesn't need to be a power of *2 (256, 512, ...)*, it can be any positive integer (say *1,000, 687, ...*)
[2] If you enter a positive integer value *P*, the intermediate results will be displayed every *P* points as well as the final result once all *N* points have been generated. *P* doesn't need to divide evenly into *N*, the final result will be displayed regardless. If *P* is *0* no intermediate results will be shown, which will mean faster execution but you won't be able to monitor progress.

Further Considerations

To choose the number of points *N* and max. iterations *K*, we'll take into account the following considerations:

- Both the correctness of the estimated area and the running time depend on *N* and *K*, the larger the better as far as the estimated area is concerned but the longer the running time will be. Also, whether you're using a physical *HP-71B* or a virtual (emulated/simulated) *HP-71B* and its underlying *OS* (*iOS*, *Android*, *Windows*, *Mac*, other) and hardware, all of it will greatly influence the choice of calculation parameters.

  Generally, a physical original *HP-71B* will be the slowest by far, and this will limit the running times allowable, probably 1-2 days at most. Some experimentation will be required, starting at a low value of *N*, *K* (say *N = 1,000* and *K = 256*) and noting the running time. Then it's possible to select how big *N* and *K* should be, as the time will be proportional to both.

- On the other hand, a virtual *HP-71B* will be orders of magnitude faster. For instance, using the *go71b* emulator on an *Android* mid-range *Samsung* tablet (as done below) will generate and check about *50* points per second at *256* max. iterations per point. Using a faster version of *go71b* and/or a faster emulator/*OS*/hardware combination (such as *Emu71 DOS* running on 32-bit *Windows XP* or *Emu71 Windows* running on 64-bit *Windows 7*, say) can get results much faster.

- Increasing the number of iterations *K* will always *reduce* the estimated area because performing more iterations weeds out points that never escaped to infinity when using *K* iterations, and thus were included in the count, but actually *did* escape when using more iterations and so weren't included now.

- However, increasing the number of points *N* while leaving *K* fixed results in estimated areas which overshoot/undershoot the area, slowly converging to the correct value of the area *for that number of iterations*, $\mathbf{M}_K$, *not* to the correct area of **M**, which would be the value for *infinite* iterations.

- This can be remedied by using a *correction factor,* which uses $K_{i,j}$ to extrapolate K_∞ as we'll see below.

Sample runs

Let's see several examples. We'll asume **STD** display mode for all results that follow.

Example 1

For starters, let's estimate **M**'s area using *5,000* points and max. *256* iterations, showing just the final result.

```
RUN  #Points,Iters,Every=  5000,256,0  END LINE
```

```
→  1397 #M-points, Area: 1.578610   { the final tally: 5,000 points generated, 1,397 landed in M }
                                    { estimated area of M, ~2 correct digits, err=4.78%,   139" }
```

Example 2

Let's improve the estimation using *2,000* points but max. *512* iterations, showing results every *500* points.

```
RUN  #Points,Iters,Every=  2000,512,500  END LINE
```

```
→       500    127     1.435100        { 13" }
→      1000    263     1.485950        { 35" }
→      1500    409     1.540567        { 67" }
→      2000    540     1.525500        { 93" }

→  540 #M-points, Area: 1.525500   { the final tally: 2,000 points generated, 540 landed in M }
                                   { estimated area of M, ~2 correct digits, err=1.26%,   93" }
```

The Ultimate Run

Now for the real McCoy. Taking the above considerations into account and as I'll be using a virtual *HP-71B* (***go71b*** for *Android*) running on a mid-range *Samsung* tablet, I'll use *300,000* points and a low *256* iterations for speed but I'll also compute and apply a *correction factor* to try and increase the precision. I'll compute this factor first, using *15X* fewer points than the main run (just *20,000*) but *4X* more iterations (*1,024*), as follows:

$$f_{corr} = Area_{20000,1024} / Area_{20000,256}$$

where $Area_{N,K}$ means computing the area using *N* points and *K* iterations. Let's proceed to compute f_{corr} :

RUN ***#Points,Iters,Every=*** `20000,256,0` **END LINE** *{ we'll show just the final result }*

→ ***5390 #M-points, Area: 1.522675*** *{ $Area_{20000,256}$ = 1.522675, 9'19" }*

RUN ***#Points,Iters,Every=*** `20000,1024,5000` **END LINE** *{ show results every 5,000 points }*

```
→      5000   1389    1.569570    { 6'56" }
→     10000   2712    1.532280    { 13'48" }
→     15000   4012    1.511187    { 20'42" }
→     20000   5350    1.511375    { 27'36" }
```

→ ***5350 #M-points, Area: 1.511375*** *{ $Area_{20000,1024}$ = 1.511375, 27'36" }*

So the *correction factor* will be: `1.511375 / 1.522675` **END LINE** → ***0.992578849722***

Now it's time for the the lengthy main computation, to which we'll afterwards apply the just calculated *correction factor*. This will take about *2h 20'* in all so we'll monitor progress ...

RUN ***#Points,Iters,Every=*** `300000,256,20000` **END LINE** *{ show results every 20,000 points }*

```
→     20000    5390   1.522675    { 9'19" }
→     40000   10779   1.522534    { 18'30" }
→       ...     ...     ...
→    300000   80601   1.517986    { 2h'20' }
```

→ ***80601 #M-points, Area: 1.517986*** *{ $Area_{300000,256}$ = 1.517986, err = 0.76%, 2h 20' }*

Finally, let's apply to the just computed area (which is in variable **A**) the *correction factor* previously computed:

`A * 0.992578849722` **END LINE** → ***1.50672***_030148 *vs. Förstemann's* ***1.50659***_188 }

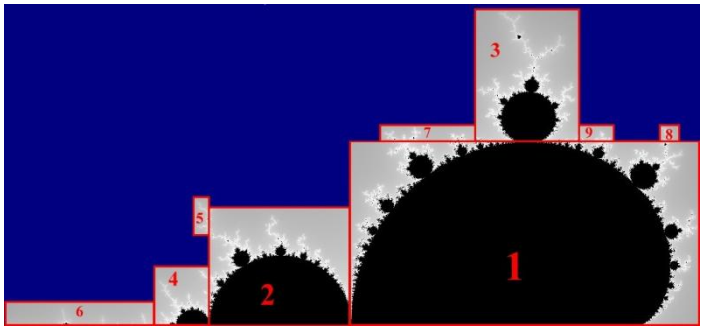
which is my final computed estimation for the area of **M** and it's correct to **5** digits within one *ulp* (unit in the last place). It differs from Förstemann's *88-trillion-pixels-calculated-at-8.6-billion-iterations-per-pixel* result by just ~ *0.000128*, an error of ~ *0.0085%*.

So he got an estimated area accurate to **9** correct digits (within possibly a couple *ulps* or three) in 35 days at great expense (both the costly hardware *and* the 35-day electricity bill), while I got **5** correct digits in less than *3 h* in all (*correction factor* computation included) at negligible expense, so point made. Not bad, isn't it ?

Where to go now

As this is an informal article and the point has already been made, we could really call it a day and move on. But if we were willing to, there's a number of further techniques to consider in order to improve the accuracy and/or reduce the computation times. For instance, among other possibilities:

- We can avoid wasting time generating and checking random points in *blank* areas (*~75%* of the enclosing box used here) where no part of **M** is, by subdividing **M** into a number of rectangular boxes (9 in the sample partition below) and then computing the total count as the sum of the counts in each individual box.



It is important to distribute the total number of points **N** among the boxes proportionally to the area of each box so that the density of points is the same. Otherwise we would be adding areas computed with different precisions and this is wasteful as the resulting sum will be no more accurate than the least accurate area.
To implement this, the program must be converted into a *subprogram* with no prompting and no output, which accepts the dimension of each box and the number of points N_i to use and returns the count to a main program which first inputs the number of points *N* and max. iterations *K* from the user and then calls the subprogram with the coordinates and the N_i for each box, then adds up the returned counts and computes and outputs the total area. There's no overhead and large blank areas are thus avoided.

Also, the process is faster for each box because some time-consuming checks are avoided altogether:

  - Box ***1*** only needs to check if points belong to the main *cardioid*, but forfeits the check for the disk.
  - Box ***2*** only needs to check if points belong to the main *disk*, but forfeits the check for the cardioid.
  - all remaining boxes forfeit **both** checks, which significantly speeds the process.

- The *correction factor* could be improved like this: we'll choose a suitable number of random points *N* and we'll obtain the count of the points belonging to **M** for an increasing max. number of iterations, say for *K* = *256*, *512*, *1024*, *2048*, etc.. We'll then analyze the counts obtained and roughly extrapolate what the expected count would be for $K = \infty$. The resulting correction factor would then be:

$$f_{corr} = count_{\infty} / count_{256}$$

which will presumably get us a more accurate estimation. For instance, for *N* = *100,000* points we get:

K	*256*	*512*	*1024*	*2,048*	*4,096*	*8,192*	∞
count $_K$	*25,501*	*25,352*	*25,312*	*25,277*	*25,261*	*25,254*	***?***

Now we simply use some extrapolation or curve fitting technique to try and estimate $count_{\infty}$.

- We can use *periodicity checking* within the iterations to detect loops and abort the iterations early.
- We can add a check for the *secondary disk* (the one in box ***3*** in the partition above) or even other μ-atoms.
- And so on and so forth ... and what about the area of *other* fractals (*Mandelbar*, *Burning Ship*, ...) ?

Notes

1. Quoting D. Allingham (see *References* below): *"This method [Monte Carlo] was employed using Mathematica, and after 20 hours and nearly 45,000 points being generated, the approximate area of the Mandelbrot set was found to be 1.4880 to 4 decimal places."* Actually the result barely has 2 correct digits and shows the amazing progress made in the last 25 years, as now I've used an inexpensive tablet to run my virtual *HP* calculator's 9-line *BASIC* program to calculate ~ 7x more points ~ 7x faster and got a result ~ 145x more accurate.

2. On the other hand, due to modern hardware (2018), the *HP-71B* emulator used here runs up to 128x faster than a physical original *HP-71B* (1984), so if using the physical model all times given here should be increased 128x, i.e.: the main result obtained in *~3 h* would take about *two weeks* or, conversely, proportionally less points must be used.

3. I've also written a 98-step (199-byte) *RPN* version of this program for the ***HP42S***. Although the random number generator is the same as the one the *HP-71B* uses, producing the exact same sequence of random numbers when using the same seed (verified up to 100 million consecutive random numbers when starting from the seed ***1***, as used in the program featured here), internally the *HP-71B* uses 15 digits (12 digits available to the user) while *Free42*, the *HP42S* emulator used, has 34-digit accuracy, which over many generated points and iterations tends to produce slightly different results, so the sample and main runs given here might not produce the exact same results.

References

Daniel Bittner *et al* (2014)	*New Approximations for the area of the Mandelbrot Set*
Thorsten Förstemann (2012)	*Numerical estimation of the area of the Mandelbrot set*
Kerry Mitchel (2001)	*A Statistical Investigation of the Area of the Mandelbrot Set*
David Allingham (1995)	*Conformal Mappings and the Area of the Mandelbrot Set*
John Ewing (1993)	*Can We See the Mandelbrot Set ?*
Ewing and Schober (1990)	*On the coefficients of the mapping to the exterior of the Mandelbrot set*
A.K. Dewdney (1985)	*Computer Recreations (Scientific American, August 1985 issue)*
Thomas Okken	*Free42: An HP-42S Calculator Simulator (website)*

Copyrights