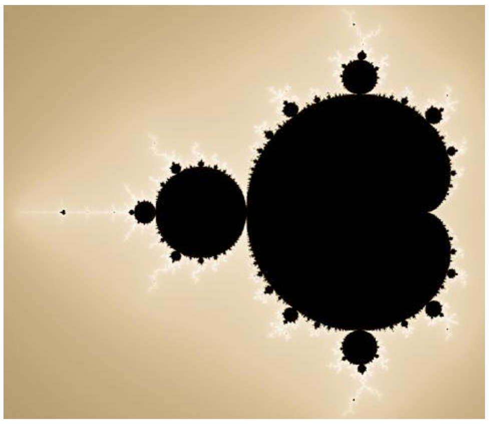
Boldly Going - Mandelbrot Set Area

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:

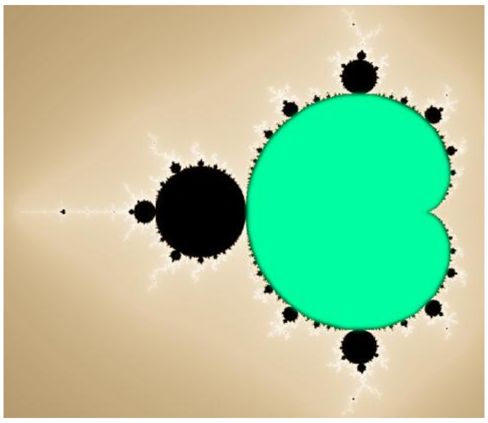
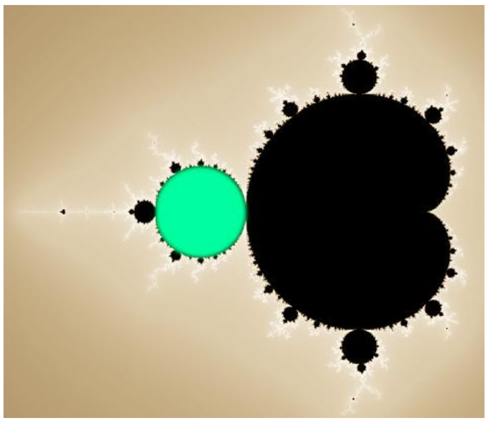
- Each point *(x,y)* will be generated as a random complex number *z* within a rectangular box enclosing **M.** Actually, the leftmost extreme of **M** is at $x = -2$, the righmost extreme is at $x = 0.471185334933396+$, the topmost extreme is at $y = 1.122757063632597+$ and the downmost extreme is at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

- 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 HP42S[1]

01	**LBL "AM"**	*26*	CF 21	*51*	ABS	*76*	GTO 00
	2.5		"Working…"		X<Y?		**LBL 03**
	STO 06		AVIEW		GTO 04		RCL 00
	2		CF 00		SIGN		RCL 03
05	STO 07	*30*	X≠0?	*55*	RCL+ ST Z	*80*	MOD
	1.2		SF 00		ABS		X≠0?
	STO 08		**LBL 00**		RCL 09		RTN
	0.25		RCL 05		X>Y?		CLA
	STO 09		STO 01		GTO 04		RCL 04
10	1	*35*	FS? 00	*60*	R↑	*85*	RCL- 00
	SEED		***XEQ 03***		RCL 07		X=0?
	"Points?"		RAN		RCL ST Y		RTN
	PROMPT		RCLx 06		**LBL 01**		AIP
	STO 04		RCL- 07		X↑2		├"→"
15	STO 00	*40*	RAN	*65*	RCL+ ST Z	*90*	RCL 02
	256		RCLx 08		ABS		AIP
	"Iters?"		COMPLEX		X≥Y?		RCL÷ ST Y
	PROMPT		ENTER		GTO 02		6
	STO 05		ENTER		X<> ST L		x
20	CLX	*45*	SIGN	*70*	DSE 01	*95*	├"$_{LF}$Area~"
	STO 02		RCL- 07		GTO 01		ARCL ST X
	"Every?"		RCLx ST L		**LBL 04**		AVIEW
	PROMPT		ABS		ISG 02	*98*	**END**
	STO 03		RCLx 09		**LBL 02**		
25	RECT	*50*	X<>Y	*75*	DSE 00		

Uses:

- *98 steps (199 bytes)*
- *flags 00, 21*
- *labels 00-04*
- *registers 00-09*
- *sets RECT mode*
- *any angular mode*

Registers:

00:	*N-loop index*
01:	*K-loop index*
02:	*M (count)*
03:	*every P*
04:	*N (# points)*
05:	*K (# iterations)*
06:	*2.5*
07:	*2*
08:	*1.2*
09:	*0.25*

Program details

Steps 01-31: main entry point: initialization[2] and prompting input from the user. *{ 31 steps }*
Steps 32-36: start of the main loop. *{ 5 steps }*
Steps 37-44: generation of a random point within the box, plus 2 copies on the stack. *{ 8 steps }*
Steps 45-53: checking whether the point belongs to the main cardioid (thus, to **M**). *{ 9 steps }*
Steps 54-59: checking whether the point belongs to the main disk (thus, to **M**). *{ 6 steps }*
Steps 60-71: checking whether the point belongs elsewhere in **M** (iterations). *{ 12 steps }*
Steps 72-73: if the point does indeed belong to **M**, increment the count. *{ 2 steps }*
Steps 74-76: decrement the number of points yet to generate/check and loop until no more left. *{ 3 steps }*
Steps 77-98: output routine, displays either the intermediate results and/or the final result. *{ 22 steps }*

[1] To enter text lines use the **ALPHA** menu; ├ is the ***Append*** character and $_{LF}$ is the ***Line Feed*** character, which can be found at the end of the second row of the **PUNC** submenu of the **ALPHA** menu.
[2] The initialization part stores four small constants in storage registers R_{06}-R_{09} because of speed considerations. Simply having the constants as program lines and performing the relevant arithmetic operations takes two program steps each and is much slower than using recall arithmetic, which just takes a single step and is faster as well. As these operations are part of the main loop, every speed gain is essential when being repeated many thousands of times.
Also, to save a register and a program step the constant ***2*** is stored just in R_{07}, then used at 3 different locations in the program, but the very first use at *step 39* depends on the enclosing box ***x***-range being from *-2* to *0.5*. If using a different box ***x***-range this constant might change and would need to be stored in its own register, say R_{10}, the other instances remaining unaltered.

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:

```
XEQ  "AM"  →  Points?             { asks for the number of points to generate, N }
  N  R/S   →  Iters?              { asks for the max.num. of iterations[1], K. Default=256, just press R/S }
  K  R/S   →  Every?              { asks if you want to display intermediate results every P points[2];
                                    if you don't and just want the final result, simply press R/S }
  P  R/S   →  Point_P →Count_P    { the intermediate tally of points generated and resulting counts }
              Area ~ Area_P       { the intermediate estimations of the area }
                 ...
           →  Point_N →Count_N    { the final tally of points generated and resulting count }
              Area ~ Area_N       { the final estimation of the area }
```

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 *HP42S*/*DM42* or a virtual *HP42S* and its underlying *OS* (*iOS*, *Android*, *Windows*, *Mac*, *Linux*, other) and hardware, all of it will greatly influence the choice of calculation parameters.

  Generally speaking, a physical original *HP42S* will be the slowest by far, and this will limit the running times allowable without depleting the batteries, probably 1-2 days at most. The *DM42* is ~100x faster and can use an *USB* power source, so it can run the program for much longer. 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 *HP42S* will be orders of magnitude faster. For instance, using ***Free42***[3] *BCD* on an *Android* mid-range *Samsung* tablet (as done below) will generate and check about *1,000* points per second at *256* max. iterations per point. This means I can use *N = 500,000* points and *K = 256* max. iterations, say, and get the result in less than 10 min. Using a faster version of *Free42* and/or a faster emulator/*OS*/ hardware combination can easily get results even 10x or 100x 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.

[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.

[3] *Free42* is a fantastic *free* simulation of the *HP42S* created by ***Thomas Okken*** for many operating systems (*Windows*, *Mac OS*, *Android*, *iOS*, *Linux*, etc.) which also runs at the heart of ***SwissMicros*** physical *DM42* calculator. It runs many hundred times faster than a physical *HP42S* and features vastly increased available *RAM*, 34-digit *BCD* precision and much more.

Sample runs

Let’s see several examples. We’ll asume $\boxed{\texttt{FIX 05}}$ display mode for all results that follow.

Example 1

For starters, let’s estimate **M**’s area using *N = 10,000* points and *K = 256* iterations, showing just the final result.

```
XEQ   "AM"   →  Points?
10000  R/S   →  Iters?              { we’ll use 256 iters. which is the default so just press R/S }
       R/S   →  Every?              { we just want the final result so just press R/S }
       R/S   →  10000 →2572         { the final tally: 10,000 points generated, 2,572 landed in M }
                Area ~ 1.54320      { the estimated area of M, just two correct digits, err=2.43%, 11” }
```

Example 2

Let’s improve the estimation using *N = 10,000* points and *K = 512* iterations, showing results every *2,000* points.

```
XEQ   "AM"   →  Points?
10000  R/S   →  Iters?
  512  R/S   →  Every?
 2000  R/S   →  2000 →511     Area~1.53300    { the first intermediate result }
             →  4000 →1041    Area~1.56150    { the 2nd intermediate result }
             →  6000 →1561    Area~1.56100    { the 3rd intermediate result }
             →  8000 →2053    Area~1.53975    { the 4th intermediate result }
             → 10000 →2560    Area~1.53600    { final result, still 2 correct digits but err=1.95%, 19”}
```

The Ultimate Run

Now for the real McCoy. Taking the above considerations into account and as I’ll be using a virtual *HP42S* (***Free42*** *BCD* for *Android*) running on a mid-range *Samsung* tablet, I’ll use half a million 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 correction factor first, using 5x fewer points than the main run but 4x more iterations, as follows:

$$f_{corr} = Area_{100000,1024} \,/\, Area_{100000,256}$$

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

```
XEQ    "AM"   →  Points?                            { we’ll use 5x less points, just 100,000 }
100000  R/S   →  Iters?                             { we’ll use first 1,024 iterations }
  1024  R/S   →  Every?                             { we won’t be monitoring progress }
        R/S   → 100000 →25312    Area~1.51872       { the value of Area_100000,1024   [ 5’45”] }
                                 STO 10             { we store it for later use }

XEQ    "AM"   →  Points?                            { as above, still just 100,000 }
100000  R/S   →  Iters?                             { now we’ll use 256 iterations, so just press R/S }
        R/S   →  Every?                             { we won’t be monitoring progress either }
        R/S   → 100000 →25501    Area~1.53006       { the value of Area_100000,256   [ 1’58”] }
                                 STO÷ 10            { R_10 now contains the c. factor ~ 0.99258853 }
```

Now it's time for the the main computation, to which we'll afterwards apply the just calculated (and stored) *correction factor*. This will take less than 10 min. in all and we'll monitor progress ...

$\boxed{\texttt{XEQ}}$	"**AM**"	→ *Points?*		*{ we'll use the full 500,000 points }*
500000	$\boxed{\texttt{R/S}}$	→ *Iters?*		*{ we'll use 256 iterations, so just press* $\boxed{\textit{R/S}}$ *}*
	$\boxed{\texttt{R/S}}$	→ *Every?*		*{ we'll monitor progress every 100,000 points }*
100000	$\boxed{\texttt{R/S}}$	*→100000 →25501*	*Area~1.53006*	*{ the first intermediate result [1'58"] }*
		...	**...**	
		→500000 →126486	*Area~1.51783*	*{ the main result, which in itself has err ~ 0.75% before applying the correction factor [9'47"] }*

Finally, let's apply to the just computed area in the display the *correction factor* previously computed and stored:

$\boxed{\texttt{RCLx 10}}$ → ***<u>1.50658</u>*** *{ more precisely, **1.50658**_263 vs. Förstemann's **1.50659**_188 }*

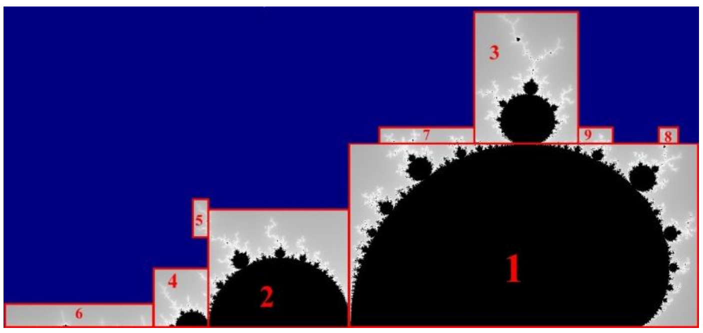
which is my final computed estimation for the area of **M** and it's correct to **6** digits within less than one *ulp* (unit in the last place). It differs from Förstemann's *88-trillion-pixels-calculated-at-8.6-billion-iterations-per-pixel* result by just ~ *0.00000925*, an error of ~ *0.000614%*.

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 **6** correct digits in less than 20 min. (actually *17'30" = 9'47"* for the main computation plus *5'45" + 1'58"* for the *correction factor* computation) 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 98-step *RPN* program to calculate ~ 10x more points ~ 60x faster and got a result ~ 10,000x more accurate.

2. I've also written a 9-line (334-byte) *BASIC* version of this *RPN* program for the ***HP-71B***. Although the random number generator is the same as the one *Free42* 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 *RPN* program featured here), internally the *HP-71B* uses 15 digits (12 digits available to the user) while *Free42* 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 shown here.

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