Very long comment to Mathologer's video: Powell's Pi Paradox: the genius 14th century Indian solution (video published 2023.05.06)

This is www.RickOstidich.com/Madhava.pdf, by Rick Ostidich, 2025.02.14

Foreword: this text was originally conceived as a comment for the YouTube video linked above; but it ended up being 10 pages long, so I decided to publish it here on my website, and on the YouTube comment I'll link to this Pdf. If you come here from a different path, please check the video first. The formula we're talking about is:

$$\pi \approx \sum_{k=1}^{n} \frac{4(-1)^{k-1}}{2k-1} + \cfrac{(-1)^n}{n + \cfrac{1^2}{4n + \cfrac{2^2}{n + \cfrac{3^2}{4n + \cdots}}}}$$

This is another one of my all-time favorite videos on YouTube.

After watching it again (for the n[th] time!), I decided to **measure** how deep a correction term we need in order to obtain the number of digits of precision required, in the fastest way; that is: **what is the lowest count of total divisions**, including both those in the alternating series and those in the continued fraction.

I worked an entire month of my free time on it, and the final result of my experimentations is **pretty impressive**: by using a slightly improved version of **Mādhava**'s formula (the same that he probably would have used in practice), **it's enough to do *n* total divisions to obtain more than *n* digits (in base 10) of precision.**

(Later in this text, I copy here all the **Pari/GP** functions that let you experiment by your own the same results, and more.)

For example: with 28 divisions you obtain **30** digits (base 10) of precision, with 94 divisions you get **100** digits, with 945 divisions you get **1000** digits.

The count of divisions is **linearly dependent** on the number of digits required. In the limit (I tested up to 1 million digits!), in order to obtain ***d*** digits (base 10) of precision, you need around ***c***=d·**0.944646…** total divisions: sum the first ***n***=d·**0.823006…** terms (which requires only 1 division for each **pair** of terms in the improved version), and add a correction term of ***m***=d·**0.533143…** divisions in the continued fraction (1 division per… each fraction).

For big-enough numbers of digits, the range for the (***n***,***m***) required to obtain the lowest ***c*** is quite wide, so that you're **guaranteed** to obtain **at least** the precision that you need by using values slightly greater than the ratios above.

Last year, **Matt Parker**'s team (Stand-up Maths channel) exerted **10'000** long divisions in order to achieve **π** with a mere **139** digits of precision.

With Mādhava's (improved) method, it's enough to do **132** long divisions (plus 131 sums, 1 shift and 1 subtraction) to achieve **140** digits of precision.

We all know that there are series which converge even more rapidly (like **Chudnovsky** algorithm), but probably Mādhava's ancient method (with the following improvements) is still the fastest for human computing.

If somebody among you happens to be in contact with **Matt Parker**, please let him know about this, so that maybe next **March 14[th]** he will use this method to manually calculate **π** to a precision humans never reached so far.

Let's start and define Madhava(n,m) as the function which returns the **π** approximation from ***n*** terms of the alternating sum, and ***m*** division in the correction term. Please remember these ***n*** and ***m*** variable names, which are used for the rest of this text.

First of all, here is a simple **Pari/GP** script that replicates **Burkard**'s example at the beginning of the video (it sums the first million reciprocals, without any correction term):

```
localprec(60); my( show(s,x)=printf("%16s = %.59f...\n",s,x) ); show("Pi",Pi); \
  x=sum(k=1,10^6, if(k%2,+4.,-4.)/(2*k-1)); show("Madhava(10^6,0)",x); \
  show("difference",Pi-x)
```

(I guess that many of you already know the free Pari/GP calculator, available for every PC, and even on "smartphones" with PariDroid. You need a recent version to use the *localprec* command, otherwise use *default(realprecision,d)*.)

The result displayed (in half a second on my old laptop) is:

```
              Pi = 3.14159265358979323846264338327950288419716939937510582097494...
 Madhava(10^6,0) = 3.14159165358979323871264338327919038419717035250010581556479...
      difference = 0.00000099999999999975000000000312499999999046875000005410 16...
```

Let's now include the correction term of ***m*** divisions, in a (simple and un-optimized) version of a complete Pari/GP function:

```
Madhava(n,m) = { my(p=0,q=4,f); forstep(k=1,2*n,2, p+=q/k; q=-q );
  if(m==0, return(p) ); f=if(m%2,n,4*n);
  forstep(k=m-1,1,-1, f=if(k%2,n,4*n)+k^2/f ); return(p+(-1)^n/f) }
```

(For non-experts: the **%** operator in Pari gives the remainder, and the **<<**,**>>** that I use later are binary shifts - i.e. multiplication by 2^n.)

Note that here we use the exact *t_FRAC* number type (a/b with big integers) instead of floating-point numbers, because they are much faster in this application. To achieve ***d*** digits of precision, the size of the integers reaches around ***d*** digits, but only towards the end of the computations (both for AltSum and ContFrac).

(**Matt Parker**: I would give it a try with integer ratios instead of floating-point, also for manual computations!)

If you want to see the decimal approximation of **π** given by this function, use "Madhava(n,m)+0.".

Example for **38** digits of precision (default in Pari), using 50 divisions (the next improved version will need only **36**):

```
? Madhava(27,23)
%1 = 85332925321309632304415481535547415273144 32 /
     27162313746755978078606073344165916835245 75

? Madhava(27,23)+0.
%2 = 3.14159265358979323846264338327950288 42

? Pi
%3 = 3.14159265358979323846264338327950288 42
```

The first is the exact ratio for the Mādhava approximation, the second is the rounded floating-point value, the third is the rounding of the actual **π** – identical here.

Let's improve this algorithm: first of all we extract the first term from the alternating sum, so that starting from the second term each pair of terms is of the form $-\frac{4}{4k-1}+\frac{4}{4k+1}$, which (thanks to **algebra auto-pilot**) simplifies to $-\frac{8}{16k^2-1}$; this saves a division for each pair of terms (hence theoretically requiring half of the time).

Moreover, as every Mathologer regular **should** know, consecutive perfect squares are separated by consecutive odd integers; so that instead of calculating ***a***=16k²-1 for each ***k***, it's enough do add a certain integer ***b*** (initially 3·16) to the previous value ***a*** (initially 15), and adding 2·16 to ***b*** at each iteration. I'll use a similar logic also for the squares in the continued fraction – which go backwards since we have to start from the bottom of the continued fraction.

Also for the second part (regarding the continued fraction), I have paired each couple of divisions in a single cycle (excluding the first term), in order to simplify the inner loop. Here too we could save a division, but at the cost of several multiplications and complicating the script - for the time being I didn't find anything better, so I decided to keep it simple.

It would be easy to change the function in order to support even numbers for ***m***, but all the tests that I did so far used this version, so we'll use this version for the rest of this text. (Otherwise, it would require me other days of work and redoing the tests, before publishing this.)

Here is the improved function (**Mādhava release 2.0**), that requires odd ***n***, and ***m*** odd or =0 (here I add the check for input parameters, assuming however that they are given as integers):

```
Madhava2(n,m) = { if(n<=0 || n%2==0 || m<0 || (m<>0 && m%2==0),
  error("invalid arguments") ); my(p=0, a=15, b=48, c=n>>1);
  while(c, p+=1/a; a+=b; b+=32; c--); p=4-p<<3; if(m==0, return(p) );
  my(f=n, n4=n<<2, a=(m-1)^2, b=2*m-3);
  while(a, f=n4+a/f; a-=b; b-=2; f=n+a/f; a-=b; b-=2); return(p-1/f) }
```

Of course, this function gives exactly the same results as the previous version, but for the example above of **38** digits it needs only **36** divisions (26/2 for the AltSum, 23 for the ContFrac) instead of **50**.

Oddly enough, in Pari/GP this function doesn't save a lot of time with respect to the previous version, but I assure you that if you implement this new algorithm in a real computer **program** (in machine language, obviously), the improvements are great. And, we are mainly interested in counting the number of divisions, which are the slowest calculations both on a computer and by hand: with the new version we save **a lot** of them.

The amazing thing is that now we can go and search for each ***n*** what ***m*** permits to obtain a certain count ***d*** of digits of precision, with the least ***c*** number of total cycles (=divisions, I use ***c*** here since ***d*** is already used for "digits"), or what ***n*** for each ***m***; then measure what are the best estimations for ***n*** and ***m*** as functions of ***d***.

Note that there is a range for (***n***,***m***) for which you obtain the same ***d*** digits with the same ***c*** cycles; for ***d***=38 digits (like in the previous example) and odd (***n***,***m***), you can use (27,23), (31,21), or (35,19), all of which require 36 ***c***=cycles=divisions: ***c***=(n-1)/2+m, and a deeper correction term give precision more quickly than more terms of the sum, within certain limits.

For big ***d***, the range is much wider, and it can be very nicely approximated (excluding exceptions) by $range \approx \frac{1}{2}\sqrt{d}$. More on this later.

As already stated above, I've seen that the optimal values of (***n***,***m***) for ***d*** digits tend to be centered around (d·0.823006,d·0.533143), so here is another function that does it all by itself according to the number of digits required, and it also shows a progression bar during the calculation - which is nice for long ones:

```
MadhavaD(d) = { my(n=bitor(round(d*0.823006),1), m=bitor(round(d*0.533143),1),
  divs=n>>1+m, p=0, a=15, b=48, c=n>>1, ch=c\50, cl=ch);
  printf("using Madhava2(%u,%u), which makes %u total divisions\n",n,m,divs);
  gettime(); print1("AltSum: ");
  while(c, p+=1/a; a+=b; b+=32; cl--; if(cl==0,cl=ch;print1(".") ); c--); p=4-p<<3;
  print(" ",strtime(gettime())); if(m==0, return(p) );
  my(f=n, n4=n<<2, a=(m-1)^2, b=2*m-3); ch=m>>1\50; cl=ch; print1("ConFra: ");
  while(a, f=n4+a/f; a-=b; b-=2; f=n+a/f; a-=b; b-=2; cl--;
  if(cl==0,cl=ch;print1(".") ) ); print(" ",strtime(gettime())); return(p-1/f) }
```

Example for 100'000 digits:

```
? x=MadhavaD(100000);
using Madhava2(82301,53315), which makes 94465 total divisions
AltSum: ................................................ 2'125 ms
ConFra: ................................................ 2'047 ms
time = 4'250 ms.

? localprec(100018); x-Pi
%2 = -3.99854867151786901316938982301460383392 E-100001
```

As you can see, here it took around **4 seconds**, and gave around **100'000** digits of precision, as expected. Pretty nice.

If you want to try MadhavaD(10^6): it does Madhava2(823'007,533'143) and gives **1 million** digits of precision, with 944'646 divisions, in around **7 minutes** of time in Pari/GP.

By the way: I also tested another famous formula variation from Mādhava, the one starting with $\sqrt{12} + \sum$ ..., but as far as I managed to simplify it, it still requires more divisions than this version.

This comment is already **too** long. If you're interested, I'm adding a comment to this comment (*p.s.: it follows here on the next page*), with all the Pari/GP functions that let you play with the formulas, and experiment all the results I found, plus surely much more if you have time to work on it: for example in order to get more precise constants, and for other numerical bases. (Personally, in most of my time, I use base G "hex" or base C "dozenal", which I like much better than the boring base A=9+1 "decimal".)

There is also a function ready to send a big table of data to a spreadsheet program, in order to draw really interesting and wonderful graphs.

And, as a homework for the keen among you, you can try and give a bullet-proof… proof that the constants I approximated here actually converge to an exact nice value, in a closed form involving **π** (or maybe **γ**). ☺

Second comment

The Pari/GP functions that automatically find the best range for (***n***,***m***) required to achieve the lowest count ***c*** of cycles (=divisions) for the count ***d*** of (base 10) digits of precision are:

```
BestN(d,m,v=1,ratc=0.944646,pi=0)= { local(tests=0, gobackn=1, x,c,minc,minn,
  n=bitor(max(round((d*ratc-m)<<1),1),1) );
  local(r(x)= round(x*10^(d-1)) ); if(pi==0, localprec(d+1);pi=r(Pi);localprec(19) );
  local(prec(x)= x=r(x); if(x==pi, d, d-#Str(abs(x-pi)) ) );
  local(testn(s)= x=Madhava2(n,m); tests++; c=n>>1+m;
    if(v>0, printf("%7s n, [c,n,m,digits~]=%5u\n",s,[c,n,m,prec(x)]) ) );
  testn("first"); if(r(x)<>pi, if(v>0, print("  - not enough precision") ); gobackn=0;
  until(r(x)==pi, n+=2; testn("inc") ) ); minc=c; minn=n;
  if(gobackn, if(v>0, print("  - go back") ); n=minn-2;
    while(n>=1, testn("dec"); if(r(x)<>pi, break); minn=n; minc=c; n-=2 ) );
  if(v>0, printf("  ;[c,n,tests]=%5u\n",[minc,minn,tests]) ); [minc,minn,tests] }

BestM(d,v=1,adapt=1,ratc=0.944646,ratm=0.533143)= {
  localprec(d+1); local(pi=round(Pi*10^(d-1)) ); localprec(19);
  local(tests, ttests=0, gobackm=1, ratc=ratc,minc,minn,maxn,minm,maxm,m1,m1r,
    m=round(d*ratm) ); if(m<>0,m=bitor(m,1) );
  local(testm(s)= [c,n,tests]=BestN(d,m,v-1,ratc,pi); ttests+=tests;
    if(adapt,ratc=c/d);
    if(v>0, printf("%5s m, [c,n,m]=%5u, tests: %u\n",s,[c,n,m],tests) ) );
  testm("first"); minc=c; minn=maxn=n; minm=maxm=m1=m; m1r=ratc;
  while(1, m+=2; testm("inc"); if(c>minc, break);
    if(c<minc, if(v>0, print("- found better above") );
      gobackm=0; minc=c; maxn=n; minm=m );
    minn=n; maxm=m );
  if(gobackm, if(v>0, print("- go back") ); m=m1-2; ratc=m1r;
    while(m>=1, testm("dec"); if(c>minc, break);
    if(c<minc, if(v>0, print("- found better below") ); minc=c; minn=n; maxm=m );
    maxn=n; minm=m; m-=2 ) );
  if(v>0, printf(";[c,minn,maxm,maxn,minm,tests]=%5u\n",
    [minc,minn,maxm,maxn,minm,ttests]) ); [minc, minn,maxm, maxn,minm, ttests] }

Table(from,to,step=1,v=1,adapt=1,ratc=0.944646,ratm=0.533143)= {
  my(ratm=ratm,c,n1,m1,n2,m2,rat,tests,tot=0);
  print("; digits  cycles       c/d  [ minn  maxm]÷[ maxn  minm]  avg(m)/d  tests");
  forstep(d=from,to,step, [c,n1,m1,n2,m2,tests]=BestM(d,v-1,adapt,ratc,ratm);
    tot+=tests; rat=(m1+m2)/2/d; if(adapt,ratm=rat);
    if(v>0, printf("%8u %7u  %.6f  [%5u %5u]÷[%5u %5u]  %8.6f %6u\n",
      d,c,c/d,n1,m1,n2,m2,rat,tests) ) );
  print("; total tests: ",tot) }

Griglia(from,to,step=1,ratc=0.944646,ratm=0.533143)= {
  print("digits,divisions,minn,maxm,maxn,minm,tests");
  forstep(d=from,to,step, [c,n1,m1,n2,m2,tests]=BestM(d,0,1,ratc,ratm);
  ratm=(m1+m2)/2/d; printsep(",",d,c,n1,m1,n2,m2,tests) ) }
```

BestN(d,m,<optional parms>) gives the best ***n*** for specific ***d***,***m***, by testing with the Madhava2 function.

BestM(d,<optional>) uses BestN to find the best range of (***n***,***m***).

Table() and Griglia() use BestM for a certain sequence of ***d*** values; Table writes the data in a nice form, Griglia does it in a delimited text ready to be pasted in a spreadsheet. Use the history file generated by Pari to copy large lists.

Every programmer among you will be able to quickly understand the optional arguments, and how the functions work in details. In Pari/GP, *function(x,y=something)* defines parameter ***y*** as optional, and assigns ***y***=<something> if the argument is not specified.

Examples (note that here I intentionally use an imprecise ***ratc*** to show an example of more work done by BestN, instead of the usual 2 tests required by an optimal ***ratc***):

```
? BestN(10000,5333,1,0.9443)
  first n, [c,n,m,digits~]=[ 9443, 8221, 5333, 9997]
  - not enough precision
    inc n, [c,n,m,digits~]=[ 9444, 8223, 5333, 9998]
    inc n, [c,n,m,digits~]=[ 9445, 8225, 5333, 9999]
    inc n, [c,n,m,digits~]=[ 9446, 8227, 5333,10000]
  ;[c,n,tests]=[ 9446, 8227,    4]
time = 267 ms.
%1 = [9446, 8227, 4]

? BestN(10000,5333,1,0.9448)
  first n, [c,n,m,digits~]=[ 9448, 8231, 5333,10000]
  - go back
    dec n, [c,n,m,digits~]=[ 9447, 8229, 5333,10000]
    dec n, [c,n,m,digits~]=[ 9446, 8227, 5333,10000]
    dec n, [c,n,m,digits~]=[ 9445, 8225, 5333, 9999]
  ;[c,n,tests]=[ 9446, 8227,    4]
time = 281 ms.
%2 = [9446, 8227, 4]

? BestM(10000)
first m, [c,n,m]=[ 9446, 8231, 5331], tests: 2
  inc m, [c,n,m]=[ 9446, 8227, 5333], tests: 2
  inc m, [c,n,m]=[ 9446, 8223, 5335], tests: 2
  inc m, [c,n,m]=[ 9446, 8219, 5337], tests: 2
  inc m, [c,n,m]=[ 9446, 8215, 5339], tests: 2
  inc m, [c,n,m]=[ 9447, 8213, 5341], tests: 2
- go back
  dec m, [c,n,m]=[ 9446, 8235, 5329], tests: 2
  dec m, [c,n,m]=[ 9446, 8239, 5327], tests: 2
  dec m, [c,n,m]=[ 9446, 8243, 5325], tests: 2
  dec m, [c,n,m]=[ 9446, 8247, 5323], tests: 2
  dec m, [c,n,m]=[ 9447, 8253, 5321], tests: 2
;[c,minn,maxm,maxn,minm,tests]=[ 9446, 8215, 5339, 8247, 5323,   22]
time = 1'422 ms.
%3 = [9446, 8215, 5339, 8247, 5323, 22]

? Table(10,90,10)
; digits  cycles       c/d  [ minn  maxm]÷[ maxn  minm]  avg(m)/d  tests
      10      10  1.000000  [   11     5]÷[   11     5]  0.500000     11
      20      19  0.950000  [   17    11]÷[   17    11]  0.550000      6
      30      28  0.933333  [   23    17]÷[   27    15]  0.533333      8
      40      38  0.950000  [   31    23]÷[   39    19]  0.525000     10
      50      47  0.940000  [   37    29]÷[   45    25]  0.540000     10
      60      56  0.933333  [   47    33]÷[   51    31]  0.533333      8
      70      66  0.942857  [   51    41]÷[   63    35]  0.542857     12
      80      75  0.937500  [   65    43]÷[   65    43]  0.537500      6
      90      86  0.955556  [   71    51]÷[   83    45]  0.533333     12
; total tests: 83
time = 16 ms.
```

```
? Table(100,900,100)
; digits  cycles       c/d  [ minn  maxm]÷[ maxn  minm]  avg(m)/d  tests
     100      94  0.940000  [   79    55]÷[   83    53]  0.540000      8
     200     189  0.945000  [  161   109]÷[  169   105]  0.535000     10
     300     283  0.943333  [  237   165]÷[  257   155]  0.533333     16
     400     377  0.942500  [  329   213]÷[  329   213]  0.532500      7
     500     472  0.944000  [  399   273]÷[  427   259]  0.532000     20
     600     567  0.945000  [  469   333]÷[  521   307]  0.533333     32
     700     662  0.945714  [  563   381]÷[  591   367]  0.534286     20
     800     756  0.945000  [  635   439]÷[  687   413]  0.532500     32
     900     850  0.944444  [  719   491]÷[  767   467]  0.532222     30
; total tests: 175
time = 297 ms.

? Table(1000,10000,1000)
; digits  cycles       c/d  [ minn  maxm]÷[ maxn  minm]  avg(m)/d  tests
    1000     945  0.945000  [  793   549]÷[  857   517]  0.533000     38
    2000    1889  0.944500  [ 1621  1079]÷[ 1673  1053]  0.533000     32
    3000    2834  0.944667  [ 2419  1625]÷[ 2523  1573]  0.533000     58
    4000    3779  0.944750  [ 3241  2159]÷[ 3345  2107]  0.533250     58
    5000    4723  0.944600  [ 4065  2691]÷[ 4169  2639]  0.533000     58
    6000    5668  0.944667  [ 4867  3235]÷[ 5011  3163]  0.533167     78
    7000    6613  0.944714  [ 5681  3773]÷[ 5841  3693]  0.533286     86
    8000    7557  0.944625  [ 6545  4285]÷[ 6625  4245]  0.533125     46
    9000    8502  0.944667  [ 7371  4817]÷[ 7443  4781]  0.533222     42
   10000    9446  0.944600  [ 8215  5339]÷[ 8247  5323]  0.533100     22
; total tests: 518
time = 14'390 ms.
```

Note that for "number of digits of precision", I mean **rounding to nearest**, and not truncation towards 0.

Since **π**=3.141'**59**2'653…, when rounded to 5 digits of precision it's better approximated by 3.141**6** than 3.141**5**.

As a matter of fact, it's curious to note that for calculating **π** to 9 digits of precision **8** divisions are enough, while for calculating it to 8 digits (one less) we need **9** divisions (one more!): Madhava2(7,5) = 3.141'592'**649**'399… which rounds correctly to 3.141'592'**65**, but incorrectly to 3.141'592'**6**. (It should be …**7**.)

At the beginning of this text, I've said that the number of division required is **less** that the number of digits requested, but this is always true only for ***d***>31.

The only ***d*** for which ***c***>***d*** (it's ***c***=***d***+1 in these cases) are **4** and **8**; the only ***d*** for which ***c***=***d*** are 1,2,3,5,7,10,31. All small numbers, that we can ignore.

And after that, you can see that the ratios ***c***/***d*** and (central ***m***)/***d*** converge quite rapidly to the constants I use, especially for big ***d***.

These are great functions to **graph**; I cannot paste pictures or documents here, so you can try it for yourself, for some very nice "a-ha" moments! *(Now I could paste the pictures in the pdf, but I don't want to spoil the joy for you.)*

About these functions, I like to point out: the method to find the "valley" of a discrete function (searching in both directions to ensure it is growing from there), the use of local sub-functions, the "verbose-level" parameter ***v*** (0—3), and the "adaptive" method to speed-up the calculations a lot when you don't know yet the precise constants to use (this has been **very** useful to quickly find the current constants).

I don't have any idea if my methods are common or not amongst programmers: I program computers since 1977, but I'm self-taught on everything, and I've never read any book nor followed any channel about "computer programming". Most of the times, I re-invent a lot of things that happen to be already invented; this is what I always liked to do, in my own ways, and several times it happened that I actually invented brand-new things. The fact that every time I face a new topic I like to invent my own method to solve it, long before looking at what is already known in the world (and picking cheap ready-made methods from the shelf), sometimes really generates methods that are much more efficient than the known ones, and more suitable for the specific purpose.

In the past, this also happened on some important **Math** topics, that I'd very like to discuss together with **Burkard**. The work shown here might catch his attention!

By the way: in year 2003, I personally invented from scratch a **fantastic** formula for quickly calculating the same **π** that we're discussing here, involving nested square roots of 2; I quickly ran to my parents' house to show it to my father, when he picked one of his book from the library and showed me a **very** similar formula, due to François Viète in 1593. What a delusion. At least, my variation was very appropriate for computer implementation. But it was too similar to Viète's, so I dismissed that. **But**, a couple of years ago, I discovered that someone recently published a paper about my identical algorithm, showing how this is a fantastic variation of Viète's formula for a PC. Damn it!

Let's go back to our functions here; to further improve them: we could save half of the time spent, by starting from a reference ***n*** value, so that we don't need to re-evaluate the same alternating sum at every iteration (the ContFrac part depends on ***n***, but the AltSum doesn't depend on ***m***); but I already spent too much time on this topic and I wish to work on something else. ☺

As already said, the ***m*** parameter is currently required to be odd; we could also support even numbers, which would increase the number of tests in the function BestM, but that would also restrict the center of the valley, so that in the end we could achieve improved results in a similar amount of time. (All as a matter of saving 1 division in 50% of the times; I don't know if it's worth it, but for the sake of completeness I would have liked to do this.)

Also, in order to estimate ***n*** I used (d·ratc-m)·2, which is very good if you are in the valley, but it's too optimistic when you are far away (this only happens in case you're trying values $d>10^6$, in order to improve my constants). Thanks to ***adapt***, it gets fixed after the first test, but it loses a lot of time on the first iteration. We could estimate ***n*** better (according to known results in smaller numbers) and save some time.

I already mentioned that in Pari/GP these functions are always much faster if they use exact ratios instead of floating point numbers (though they require a *gcd* calculation after every operation), probably because in floating-point they require huge precision for every division. (And of course, with ratios of integers the divisions are accomplished by multiplications, which are much faster.)

In my **Rix** language (my own syntax for machine language), I would implement these computations in a much different way, without the need of most *gcd* calculations. I experimented that working on pure integers, and doing a *gcd* each 1000 operations (to periodically reduce the dimension of the integers), you save half of the time even in Pari/GP.

And, definitely we can improve the **algorithms** a lot, which as always is **much** more important that optimizing the code.

In the end: these versions of the functions are quick enough to test up to around **100'000** digits of precision.

In order to measure the constants for 1 million digits, I actually used what people call **binary search** to very quickly find the limits of the wide valley, according to the aforementioned $range \approx \frac{1}{2}\sqrt{d}$, then saving a lot of time while ensuring that no lower ***c*** is hidden somewhere within the current range.

The above functions could always exploit this technique, and save a lot of time.

As always, it's very important for me to emphasize that Pari/GP (or other computer scripts - known as "languages") are very easy and fun for experimenting the initial part of a project; but once you find the optimal algorithm, and seek real efficiency, or want to work with huge numbers and a lot of iterations, it's necessary to write a real computer **program**, in the **only** language that a computer directly understands: **Machine Language**.

If you trust my long experience, a program implemented in good machine language it's usually 100 to **1000** times faster than a version in other modern "languages", not to count the greater compactness, and most importantly: the reliability. Once upon a time, there was a saying that a program very well written in C would only be two to four times slower than in machine language, and that was probably true at the time. But things got **much** more complicated nowadays, and modern “programming languages” got definitely obscene and extremely far away from the contact with the reality of the CPU, which in the end must do the work.

But probably it's better that I talk about this elsewhere, or in another comment if you ask.

Rix: *(Okay, I included this section here, feel free to skip it if you’re not interested in it.)*

Every time I converted Pari/GP scripts to Machine Language, I got a speed increase better than 1000 times, and more importantly, even a better time complexity (big O notation).

And this is not because I'm the god of computer programmers (or maybe?!): the reason is that I still use the same programming style that it was mandatory to follow in the `70, when computers had very limited speed and memory. Plus, okay I'm quite good at it, and I never stopped improving that style since the seventies, before and after inventing my new syntax “Rix”. I also taught this programming style to **other people**, and in a few years they were achieving results similar to my own, or maybe even better. I’d be very happy if someone does better yet. And my ultimate goal is to contribute to the quality of worldwide computer programming – which is currently a disaster.

I understand that a beginner, compared to something like basic or python, might find it “difficult” to learn the complete machine language instruction set, with registers usage and stack and memory access, and without the help of **any** sophisticated structure provided by the compiler; but there is no shortcut in computer programming, like in the rest of Mathematics, and in Life in general.

And (I swear it!) when you're **no longer** a beginner it saves you **a lot** of time as you don't have to deal with all the intricacies and red-tapes required by "high-level languages". Also, you don't have to trust **anyone** but yourself.

It is very similar to the case in which if you have to speak to a **Chinese** person; you have 3 ways of doing that:

1) Learn to fluently speak the Mandarin language (including a lot of common proverbs) and correctly write all the Chinese characters; so that you have **complete** control of what you say and write, without any possibility of misunderstanding (which is essential for computer programming).
2) Hire a human translator who does that for you, who is an expert you can trust infinitely, and who has the time and ability to listen and understand every **concept** that you explain.
3) Take it easy and use Google Translate. But, don’t expect great results.

Indeed, writing in python (or even in C) is just like using a **very** sophisticated Google Translate, that in order to achieve the level of complexity required by computer programming, it gets **much much** more complicated than actually learning the actual Machine Language. **And** Chinese.

Also, if you speak a certain language, you’ve trained yourself to directly **reason** in that language, which gives you a better connection to what you’re actually trying to communicate or obtain.

The fact that nowadays it’s no longer fashionable to write Machine Language programs (we all can see the sad results) it's only a matter of business and money interests.

Personally, I don't need Borland, I don't need Microsoft, nor any other brand nor teacher nor consultant nor any of the thousands of companies that always try to convince programmers that they need their support.

The **only** thing that I actually need are the official **Intel** CPU manuals, which are available for free on their site.

When I program computers, I only need 2 program files: the text editor **Thule.exe** I wrote in 1989 (to write the source code), and the **Rix.exe** assembler I wrote in 2004 (before then, I was using the old ASM Assembly syntax) to convert the source files directly to the executable format. Together, they require 179 kB of space, including Unicode fonts and icons. **No** other programs required. No *linker*, since the executable file depends bit-by-bit by the source. Never used or desired any *debugger* or *profiler*. I **never** used any *library*, nor any piece of code written by anyone else. No *macros*, no *structures*. And my programs use the Operating System's APIs **only** for things that the OS doesn't allow me to do by myself. Hard and wild life, but very much worth it in the end.

I still have to complete the English **manual** for Rix with all of the details (it'll be an entire book about **Math** and **Language**); I hope to finish it within this year 2025, then I'll publish Rix and Thule **for free** on my website.

Note: I'm a bit conflicted by the fact that this text is not 100% complete (for example, I'm still not sure if to include this section about Rix, as I would really like to avoid such things as self-celebration and complaints); but I realize that most of the works that I did in my life are not yet published just because they're not 100% complete – as they'll probably never be. So, I decided to publish this as it is right now, then maybe I'll think about a new version later.

Conclusion, and credits:

It is very funny, and probably a bit silly, that during this long work that I did on Mādhava's formula, my computer actually calculated a very precise **π** for **billions** of times, just in order to show what is the quickest way to approximate it. ☺

But it was "quality" time, for me, and also for the CPU (I can speak to it, and **it** told me!); I enjoyed this a lot, and maybe this work will be useful (or at least interesting) for someone else.

And, it's been another continuation of the work of the great **Mādhava** of **Sangamagrāma** and disciples - whom I didn't even heard of before watching **Mathologer** videos. Another very long collaboration in Mathematics!

One reason for which I like so much **Burkard**'s videos, is that I see in him something of myself (I am a flight instructor, and in my theory lessons I've always used a similar entertainment approach), and most of all I see in him something of my father **Mario**, who since my childhood is still teaching me the passion for Mathematics and Science (being himself a self-taught person, as I am).

In a couple of Burkard's videos, I've seen from him the same respect and affection for **his** father; and in the latest video (the one about Helicones) the same from his children to him.

Hence, I'd like to dedicate this work of mine to **Polster** family, and to my father **Mario** himself! 😊 🥹

If you liked the work I did on this topic, please check the most recent comments to Burkard's other video about **Fermat's two square theorem**, where in a couple of days I'm going to add another equally interesting (but much shorter) comment, with other Pari/GP ready-to-use functions, about the results I achieved on that topic too.

Thanks for your time, ciao!

■ Rick Ostidich - made in Italy - © 1969