Mean Matrices

Valentín Albillo (#1075, PPC #4747)

We're so used to our beloved HP calculator's high-precision, high-quality math algorithms that we tend to take for granted that each and every result we get is computed accurately to 10 or 12 digits, give or take a few counts in the last places, specially if we're not making a chain calculation but just a single operation. At most, we're aware that near the range of legal values for that operation we might perhaps be too close to some singularity or peculiarity of the given function, but these are well documented, easily recognizable pathological cases that we both expect and know how to live with, or even avoid them altogether. Let's say for instance trying to use `TAN` for arguments very close to Pi/2.

Unfortunately, there are times when we neither expect nor are able to predict that some given data will completely baffle internal algorithms, resulting in grossly inaccurate results, which we, trusting them, are unable to recognize as such. This isn't due to any kind of bugs or lack of sophistication in the internals but is a consequence of the very nature of the problematic arguments themselves.

The problem

The problem can most easily be exposed for the case of some very common matrix operations, such as inverse, determinant and linear system solving, when applied to so called *ill-conditioned* matrices. A very well-known family of NxN very ill-conditioned, symmetric matrices are the **Hilbert matrices**. This is their 4x4 instance:

```
        1    1/2  1/3  1/4
  H4 = 1/2  1/3  1/4  1/5
       1/3  1/4  1/5  1/6
       1/4  1/5  1/6  1/7
```

Hilbert matrices are a favourite for the purpose of testing or comparing matrix algorithms, even among different calculator models, because of their strong ill-conditioning. However, I've always advocated that, in fact:

- They're highly ***misleading*** when used to assess some algorithm's accuracy, because you're <u>*not*</u> computing the determinant of said Hilbert matrix to begin with, but that of an *approximation* to said matrix, because terms such as 1/3 and 1/7 can't ever be represented exactly and so are internally stored with different accuracies (i.e.: different values) in different calculators. As the initial matrix being used is *not* the same, and as precisely the Hilbert matrices are *extremely ill-conditioned*, meaning that the *smallest* change in the input brings out a *large* change in the output, it's fairly obvious that <u>the results can't be meaningfully compared</u>.
- There's also the fact that Hilbert matrices, specially the large ones, are pretty ***unbalanced***, with some elements being too large when compared with others.

For instance, in the small 4x4 instance above, there's almost one denary order of magnitude between 1 and 1/7 (and almost three binary orders of magnitude). This alone does make for non-significant error accumulation. We could try to get rid of the non-exact values by multiplying all elements by some large value in order to make sure they are integer to begin with, but this would result in very large entries and would do nothing for the balance, you'd still have large values mixed with much smaller ones.

- Hilbert matrices are ***symmetrical***, which means you can't use them to check the different results you obtain from your algorithms when working with the matrix and its transpose (usually *very* different). It also means some algorithms could hypothetically make use of their symmetry to apply special-purpose routines which can mislead you as they won't work for general, non-symmetric matrices.
- There are ***ad-hoc*** algorithms that can compute the exact or approximate value of the determinant of any Hilbert matrix with great accuracy, without resorting to general matrix algorithms. You can never tell if published results have been obtained with such an algorithm, and it would be convenient if no such shortcuts were available. For instance, this small HP-71B program will compute the determinant of assorted Hilbert matrices extremely quickly and accurately:

```
 ! ** Determinants for Hilbert matrices from order 7 to 10
 FOR N=7 TO 10 @ P=1 @ FOR I=1 TO N-1 @ P=P*FACT(I) @ NEXT I @ Q=1
 FOR I=N TO 2*N-1 @ Q=Q*FACT(I) @ NEXT I @ DISP N;P^3/Q @ NEXT N

>RUN
             7  4.83580262391 E-25
             8  2.73705011379 E-33
             9  9.72023431183 E-43
            10  2.16417922642 E-53
```

However, the original idea of using some suitably large, difficult (read "ill-conditioned") matrix is inherently a good accuracy test, as it requires many arithmetic operations, most of them carried near the limits of internal accuracy, where errors are usually largely amplified by the combined effects of the finite accuracy of the initial values' internal representations *and* the choice of basic arithmetic algorithms.

What to do, then ? The best of both worlds, namely:

- Let's try and use a suitably not-too-large, difficult matrix ...
- ... but let this initial matrix be balanced and exactly representable, so that the very same matrix is actually processed for all models and algorithms, and so that the results are indeed comparable and really shed some valid light on the respective accuracies.

To that effect, I propose the following 7x7 matrices ("**Albillo Matrices**") that I've carefully crafted for this purpose, here's the #1 instance:

```
             58  71  67  36  35  19  60
             50  71  71  56  45  20  52
             64  40  84  50  51  43  69
   AM #1 =   31  28  41  54  31  18  33
             45  23  46  38  50  43  50
             41  10  28  17  33  41  46
             66  72  71  38  40  27  69
```

which is a random 7x7 matrix, so that even the HP-15C can find its determinant, (8x8 would be too large), with quite *small*, 2-digit elements, yet suitably difficult indeed. As I see it, they have a number of important advantages when compared with Hilbert matrices:

- They have small, integer elements which can be represented exactly in any calculator/computer architecture, thus guaranteeing that you start with the same, original matrix.
- All elements are 2-digit, which makes for a better test as the initial condition does not imply arithmetic between disparage values right from the start.
- They are random matrices, with no symmetry whatsoever. You can test them in their original form versus their transposes, and no special purpose algorithm applies to them.
- There are no special theoretical algorithms to deal with them. You need some pretty accurate algorithms and the best architecture possible if you want your results to be any useable at all.
- And last but not least, they are incredibly ill-conditioned, even more so than Hilbert matrices of comparable dimensions. If needed, matrices such as **AM#1** can be crafted for any specific dimensions, with an exponentially increasing intractability. A 10x10 instance will probably result in utterly worthless results when processed with most any calculator model.

We may now try it with several machines [1], let's say computing its determinant. The correct value is exactly **1**. We can also try to compute its inverse, and the inverse's determinant (also **1**). Computing the product of the matrix and its computed inverse should result in an *identity matrix*, and come to it, we can also try to solve a system of linear equations for good.

[1] A word of caution for the HP-48G and HP-49: when flag -54 is clear, then an extra check is performed before the determinant is calculated: to see whether all matrix elements are integer, for then, the result must be integer, too, and it is *changed* to the nearest integer! With **AM#1**, flag -54 clear, the result is exactly **1**. With flag -54 set, it is **.999945522778**, delivering 4 correct digits. So, set that flag -54 on your 48's and 49's for meaningful comparisons (Werner *dixit*).

The Results

These are the results you get when processing **AM #1** with an **HP-71B + Math ROM** and an **HP-15C**. As you will see, despite the "easy" look of the matrix (a small 7x7 matrix, with small 2-digit positive integer elements), even 15 digits of internal accuracy won't get us more than *two* correct digits in our results.

Less internal digits, say HP-15C's 13, won't give us even *one* correct digit in some computations. Worse, if the results are to be used as some intermediate step in a chain calculation, we'll end up having no correct digits at all pretty soon. Should you happen to inadvertently stumble upon such an "innocent-looking" matrix as this one, blindly trusting your results can mean catastrophic failure:

Now for the commented results for **AM #1**:

- **Exact Results**:

  Determinant = **1**
  Inverse =

```
 96360245   320206  -537449   2323650 -11354863  30196318 -96509411
     4480       15      -25       108      -528      1404     -4487
   -39436     -131      220      -951      4647    -12358     39497
   273240      908    -1524      6589    -32198     85625   -273663
 -1846174    -6135    10297    -44519    217549   -578534   1849032
 13150347    43699   -73346    317110  -1549606   4120912 -13170704
-96360787  -320208   537452  -2323663  11354927 -30196488  96509954
```

  Determinant of Inverse = **1**
  Product of Matrix * Inverse = **7x7 Identity Matrix**
  Row Norm of Product = **1**
  Column Norm of Product = **1**
  Frobenius Norm of Product = **2.64575131106+**

- **Computed Results for HP-71B - Method 1:**

  Determinant (with `DET`) = **0.97095056196** (error = 2.91%)
  Inverse (with `MAT INV`) = (shown truncated to integer values)

```
 99243204  329786 -553528   2393170 -11694584  31099748 -99396833
     4614      15     -25       111      -543      1446     -4621
   -40615    -134     226      -979      4786    -12727     40678
   281414     935   -1569      6786    -33161     88186   -281850
 -1901408   -6318   10605    -45850    224057   -595842   1904352
 13543786   45006  -75540    326597  -1595967   4244203 -13564752
-99243762 -329788  553531  -2393183  11694650 -31099923  99397392
```

  Determinant of Inverse (with `DET`) = **1.05907852363** (err = 5.91%)
  Product of Matrix * Inverse (with `MAT *`) = (not shown)
  Row Norm of Product (with `RNORM`) = **1.0852916** (err = 8.53%)
  Column Norm of Product (with `CNORM`) = **1.04948** (err = 4.95%)
  Frobenius Norm of Product (with `FNORM`) = **2.65606227412** (err = 0.39%)

- **Comment:**

  This is a vey difficult matrix, and though the HP-71B carries 15 digits internally a significant loss in accuracy is to be expected. So we get a **2.91%** error when calculating the determinant, which is good to *only two digits*, and an inverse which is good to *only a digit and a half*, at best. The error in the computed determinant of the inverse rises to **5.91%**, though the product of both is actually surprisingly close to the identity matrix, despite the notorious inaccuracy of the inverse. Even so, the norms of the product are in error by **8.53%**, **4.95%** and **0.39%**.

- **Computed Results for HP-71B - Method 2:**

  Inverse (with `MAT SYS`) = (shown truncated to integer values)

```
  96273474    319931 -536969  2321578 -11344786   30170494  -96423005
      4475        14     -24      107      -527       1402      -4482
    -39400      -130     219     -950      4642     -12347      39461
    272993       907   -1522     6583    -32169      85551    -273417
  -1844511     -6129   10287   -44479    217355    -578039    1847376
  13138505     43661  -73280   316827  -1548230    4117387  -13158912
 -96274016   -319933  536972 -2321591  11344850  -30170664   96423547
```

  Determinant of Inverse (with `DET`) = **1.01830872148** (err = 1.83%)
  Product of Matrix * Inverse (with `MAT *`)= (not shown)
  Row Norm of Product (with `RNORM`) = **1.0044644** (err = 0.45%)
  Column Norm of Product (with `CNORM`) = **1.00896** (err = 0.90%)
  Frobenius Norm of Product (with `FNORM`) = **2.64599734715** (err = 0.009%)

- **Comment:**

  Though the 1st method makes use of the obvious `MAT INV` matrix statement to compute the inverse, this is not the *only* way to do it nor the *best*, and the ***Owner's Handbook*** itself instructs us to use `MAT SYS` instead for extended accuracy and speed (though at the cost of extra memory being required). Doing so pays handsomely, as the inverse matrix comes out much more accurate, and its determinant is in error by a mere 1.83%, more than *3 times more accurate* than using `MAT INV` (5.91%). The product also resembles the identity matrix much more closely, and its norms show a very marked improvement when compared to those of the 1st method. This is obviously the way to go in all cases, if RAM allows.

- **Computed Results for HP-15C:**

  Determinant (with `MATRIX 9`) = **1.080204421** (err = 8.02%)
  Inverse (with `1/X`) = (not shown)
  Determinant of Inverse (with `MATRIX 9`) = **0.6635047876** (err = 33.64%)

- **Comment:**

  Despite the extremely difficult original matrix and the HP-15C internal algorithms being limited to 13 digits, the determinant is still within 8% error, i.e.: nearly *two* correct digits. However, the inverse matrix isn't that good and its determinant is already at 34% error, which means *not even one* decimal digit is correct. Due to memory limitations it isn't possible to compute the product of the original matrix and its inverse nor the corresponding norms.

What can be done ?

Though using **DET** to compute **AM #1**'s determinant does result in quite poor results (due to their extreme ill-conditioning and inherent loss of precision caused by the many division operations the internal triangularization algorithm entails), this doesn't mean we can't obtain exact results in an HP- 71B, say.

In fact, we just need another approach that doesn't involve divisions at *any* step, and one such approach is to use the ***determinant expansion by minors*** technique. This algorithm is of considerable theoretical interest, and can be recursively used to compute any determinant using just multiplications and additions, never a division in sight. However, it is *highly inefficient* for matrices above, say, 4x4. But for the sake of demonstrating how you can *in theory* compute those difficult determinants *exactly,* it will do.

Thanks to the powerful HP-71B's BASIC dialect, a simple version of the expansion by minors can be coded as a *recursive* subprogram in just 4 lines:

```
SUB XDET(A(,),D) @ DIM N,E,I,J,K @ N=UBND(A,1)
IF N=2 THEN D=A(1,1)*A(2,2)-A(1,2)*A(2,1) @ END ELSE DIM B(N-1,N-1) @ D=0
FOR K=1 TO N @ FOR I=2 TO N @ C=1 @ FOR J=1 TO N @ IF J#K THEN
   B(I-1,C)=A(I,J) @ C=C+1
NEXT J @ NEXT I @ CALL XDET(B,E) @ D=D-(-1)^K*A(1,K)*E @ NEXT K
```

You must previously dimension the matrix (with **OPTION BASE 1**) and a real variable to return the value of the determinant, then populate the matrix and call the subprogram, like this example using **AM #1.** From the keyboard do:

```
>OPTION BASE 1 @ DIM A(7,7) @ MAT INPUT A  [ENTER]

A(1,1)? 58,71,67,36,35,19,60,50,71,71,56,45,20,
        52,64,40,84,50,51,43,69 [ENTER]
A(4,1)? 31,28,41,54,31,18,33,45,23,46,38,50,43,
        50,41,10,28,17,33,41,46 [ENTER]
A(7,1)? 66,72,71,38,40,27,69    [ENTER]

>DET(A)  [ENTER]

     .97095056196  (inexact result with DET)

>CALL XDET(A,D) @ DISP D  [ENTER]

     1 (exact result with XDET)
```

Keep in mind that this version of **XDET** is just a *proof-of-concept* implementation, to show that it can be done, and as such has no error handling (the matrix must be at least 2x2, for instance), and is *extremely* inefficient as it has to recursively call itself (N-1)!/2 times to compute an NxN determinant.

This means that for **AM#1** it calls itself (7-1)!/2 = 360 times, and it will take some 2 min. in Emu71 running on a 2.4 Ghz PC, and one hour or two on a real HP-71B. It also uses up large amounts of RAM to cater for the recursion.

However, it can be easily optimized in a number of ways. For instance, a preliminary search to locate rows/columns containing 0's, then expanding by minors along these rows/columns would mean avoiding a whole sub-branch of recursion for each 0 located. If no 0's are present, they can often be created by suitably adding and subtracting rows, just remember to avoid inexact divisions.

Also, when N is 2, **XDET** refrains from recursing again to compute the required 1x1 determinants, but computes the 2x2 determinant instead directly (halving the number of recursive calls needed). This could be done for N=3 instead, which would result in much greater time savings.

Can it get any worse ?

Unfortunately, yes. A lot. As an example, we'll deal now with a much deadlier instance of my own, hand-crafted **Albillo Matrices**, this time it's **AM#7**, which is:

```
                 13  72  57  94  90  92  35
                 40  93  90  99   1  95  66
                 48  91  71  48  93  32  67
     AM #7 =      7  93  29   2  24  24   7
                 41  84  44  40  82  27  49
                  3  72   6  33  97  34   4
                 43  82  66  43  83  29  61
```

Though similar to other matrices in the suite, such as **AM#1**, this one is so extremely ill-conditioned that a ***Mathematica*** session is included to vividly demonstrate just how bad results can turn out to be.

For starters, let's compute its determinant using our trusty HP-71B (+Math ROM):

```
   >DET(A)

        0.0699243217409
```

As the exact determinant is **1**, you can see that our computed value is purely *garbage*. Computing the inverse, either with `MAT INV` or `MAT SYS,` gives a result which has *no resemblance whatsoever* to the true inverse.

The determinant of this 'inverse' gives either **+35.4852567156** (using `MAT INV`) or **-11214.4851552** (using `MAT SYS`), both hopelessly far from the actual value, **1**.

For the final *coup-de-grace*, let's try to solve the following **AM#7**-based system of linear equations:

$$\begin{array}{rrrrrrrl}
13\,x_1 & +\,72\,x_2 & +\,57\,x_3 & +\,94\,x_4 & +\,90\,x_5 & +\,92\,x_6 & +\,35\,x_7 & =\ 453 \\
40\,x_1 & +\,93\,x_2 & +\,90\,x_3 & +\,99\,x_4 & +\,x_5 & +\,95\,x_6 & +\,66\,x_7 & =\ 484 \\
48\,x_1 & +\,91\,x_2 & +\,71\,x_3 & +\,48\,x_4 & +\,93\,x_5 & +\,32\,x_6 & +\,67\,x_7 & =\ 450 \\
7\,x_1 & +\,93\,x_2 & +\,29\,x_3 & +\,2\,x_4 & +\,24\,x_5 & +\,24\,x_6 & +\,7\,x_7 & =\ 186 \\
41\,x_1 & +\,84\,x_2 & +\,44\,x_3 & +\,40\,x_4 & +\,82\,x_5 & +\,27\,x_6 & +\,49\,x_7 & =\ 367 \\
3\,x_1 & +\,72\,x_2 & +\,6\,x_3 & +\,33\,x_4 & +\,97\,x_5 & +\,34\,x_6 & +\,4\,x_7 & =\ 249 \\
43\,x_1 & +\,82\,x_2 & +\,66\,x_3 & +\,43\,x_4 & +\,83\,x_5 & +\,29\,x_6 & +\,61\,x_7 & =\ 407
\end{array}$$

which has the obvious, unique solution:

$$\underline{x_1 = x_2 = x_3 = x_4 = x_5 = x_6 = x_7 = 1}$$

Let's do it. Assuming **AM#7** has already been input as a 7x7 matrix called **A**:

```
>DIM X(7), Y(7)  [ENTER]

>MAT INPUT Y     [ENTER]

Y(1)?  453,484,450,186,367,249,407 [ENTER]

>MAT X=SYS(A,Y) @ MAT DISP X       [ENTER]

  53.464450699, 0.993385920659,  47.7783081926,  48.0455697467
  1.03246161789, -46.0448461769,  -97.428699025
```

which is an unmitigated *disaster* with *no* resemblance to the actual solution. Let's see what happens if the independent terms are altered by as little as 0.001:

```
>MAT INPUT Y  [ENTER]

Y(1)? 453.001,484.001,450.001,186.001,367.001,
      249.001,407.001  [ENTER]

>MAT X=SYS(A,Y) @ MAT DISP X  [ENTER]

 -40954525.8642, 5164.04830127, -36515838.8502, -36724467.1754
 -25339.0194654, 36723904.3462,  76834900.5317
```

which not only has no resemblance to the actual solution but also to the previous one as well. Let's now alter the independent terms by as little as 0.00001:

```
>MAT INPUT Y  [ENTER]

Y(1)?  453.00001,484.00001,450.00001,186.00001,
       367.00001,249.00001,407.00001 [ENTER]

>MAT X=SYS(A,Y) @ MAT DISP X  [ENTER]

  -409175.657315, 52.5840127453, -364828.737586, -366913.14308
  -252.172121748,  366909.499872, 767658.442598
```

which, again, resemble no actual or former solutions at all. This is exactly what ill-conditioned means: the *slightest* change in the coefficients means a *drastic*, disproportionate change in the solutions. If the coefficients are obtained by some real-life, experimental process, the computed solution will be *meaningless*.

Righting the wrongs

Just for comparison purposes, perhaps with other models implementing extended precision, a ***Mathematica*** session follows, which calculates all results using exact arithmetic and algorithms. First, we'll input **AM#7**:

```
am7 = {{13,72,57,94,90,92,35}, {40,93,90,99, 1,95,66},
       {48,91,71,48,93,32,67}, { 7,93,29, 2,24,24, 7},
       {41,84,44,40,82,27,49}, { 3,72, 6,33,97,34, 4},
       {43,82,66,43,83,29,61}}
```

Let's display AM#7 in matrix form:

```
am7//MatrixForm
    13   72   57   94   90   92   35
    40   93   90   99    1   95   66
    48   91   71   48   93   32   67
     7   93   29    2   24   24    7
    41   84   44   40   82   27   49
     3   72    6   33   97   34    4
    43   82   66   43   83   29   61
```

Let's compute its determinant and exact inverse matrix:

```
Det[am7]
   1
Inverse[am7]//MatrixForm
  71082  -507460  -2128901626  -36543896  265158513   3554051   2129774383
     -9       64       268386       4607     -33428      -448      -268496
  63378  -452461  -1898169428  -32583237  236420404   3168860   1898947595
  63740  -455046  -1909014363  -32769397  237771160   3186965   1909796976
     44     -314     -1317227     -22611     164063      2199      1317767
 -63739   455039   1908985002   32768893 -237767503  -3186916  -1909767603
-133357   952047   3994038146   68560103 -497464609  -6667765  -3995675528
```

Now for the inverse's determinant:

```
Det[Inverse[am7]]
   1
```

Let's solve the original system of equations:

```
ind1 = {453, 484, 450, 186, 367, 249, 407}
LinearSolve[am7, ind1]
     {1, 1, 1, 1, 1, 1, 1}
```

And now, the first altered system, first in exact, rational form:

```
   eps = 1/1000
   ind1 = {453 + eps, 484 + eps, 450 + eps, 186 + eps,
           367 + eps, 249 + eps, 407 + eps}
   LinearSolve[am7,ind1]
 232606047    7081   207396111  41716207  144921    208575827     436389963
{---------, -(----), ---------, --------, ------, -(---------), -(---------)}
   1000       250      1000       200      1000       1000          1000
```

Then resolving the fractions to approximate values:

```
   N[LinearSolve[am7,ind1],15]
        {232606.047, -28.324, 207396.111, 208581.035, 144.921,
        -208575.827, -436389.963}
```

Finally, the second altered system:

```
   eps = 1/100000
   ind1 = {453 + eps, 484 + eps, 450 + eps, 186 + eps,
           367 + eps, 249 + eps, 407 + eps}
   LinearSolve[am7,ind1]
 232705047  17669  207495111  41736007  243921    208476827     436290963
{---------, -----, ---------, --------, ------, -(---------), -(---------)}
  100000    25000   100000     20000    100000     100000        100000
   N[LinearSolve[am7,ind1],15]
        {2327.05047, 0.70676, 2074.95111, 2086.80035, 2.43921,
        -2084.76827, -4362.90963}
```

As you can see, even exact computing and exact algorithms will prove useless in the presence of serious ill-conditioning: the very slightest differences in some experimentally obtained values will result in completely different answers, as demonstrated by the solutions above for eps=1/1000 and 1/100000.

Homework

You might want to try some fresh examples yourself with your favourite machine or software. If so, you might want to try these **AM** instances (`DET`=**1** in all cases).

AM #2 :

```
          86  69  53  33  18  10  87
          70  65  63  55  20  25  73
          44  53  63  67  45  44  49
          32  27  50  84  36  12  33
          28  41  39  64  51  45  33
          11  20  13  17  19  33  15
          88  73  55  35  21  18  90
```

AM #3

```
60  53  61  65  50  37  64
49  78  67  67  24  10  50
31  52  59  60  32  19  33
18  51  50  82  35  27  21
10  18  30  36  38  12  11
 6  18  14  33  13  33  10
61  57  63  72  51  45  66
```

AM #7b :

```
13  72  57  94  92  90  35
40  93  90  99  95   1  66
48  91  71  48  32  93  67
 7  93  29   2  24  24   7
41  84  44  40  27  82  49
 3  72   6  33  34  97   4
43  82  66  43  29  83  61
```

This last one vividly shows that, when dealing with ill-conditioned matrices, even merely *reordering* the rows and columns can have a *great* impact on accuracy. For example, this reordering makes the HP-71B's `DET` command return the value:

-0.0611338355796

whereas the original ordering gave **0.0699243217409** instead, a 12% difference.

Conclusion & recommendations

I hope this article has enlightened you to the dangers of blindly relying on our beloved machines' internal algorithms come what may, by conclusively showing unexpected examples where they fail miserably.

You should always *test* your results, usually by repeating the computation using a different accuracy and/or algorithm, to detect any suspicious *sensitivities*. For the case of the matrix operations discussed in this article, a good telltale of latent ill-conditioning is the fact that their determinants are *far too small* for their dimensions and element-size, which places them on the verge of near-singularity.

For instance, the 7x7 Hilbert matrix' determinant is **4.83580262391E-25**, while the determinant of my **AM#1**, **#2**, **#3**, and **#7** matrices is exactly **1**. This doesn't seem that small until you consider that for a random 7x7 matrix with 2-digit integer elements the average determinant is about **2,000,000,000,000** ! This means their rows/columns are very close to being *linearly dependent*, and thus the matrices are very close to being *singular*. Solving such a system of equations means finding the intersection of nearly parallel hyperplanes, so the point of intersection will vary wildly with the precision and algorithm used.