Boldly Going ... Matrix Square Root

Valentín Albillo (HPCC #1075)

Welcome to the very first of a new series of articles under the generic name of **Boldly Going ...** (“*where no HP calc has gone before*”, of course !), where I’ll (boldly) go into math topics very rarely seen, if at all, as they’re frequently neglected due to their perceived unusual difficulty. Most people wouldn’t know where to begin in order to tackle them: neither do our calculators feature them as built-in functionality nor can programmed solutions be found. Nevertheless I’ll show, in a non-rigorous way which you’re expected to improve upon, that the beast can be tamed (usually with surprisingly little code), so that you’ll be able to do with your vintage HP things many all-powerful, newer models won’t even touch.

For instance, let’s suppose that some friend, duly impressed by your brand-new **HP48/49/50**, after hearing from you that it’ll do all sort of matrix operations, went then to ask you to find the ***matrix square root*** of these two neat little matrices:

```
      |   56   97   17   89 |      | 4 +  i   7 +  i   3 -  i   4 + 2i |
  A = |   33  -68  -42    5 | , A = | 6 -  i   9 + 4i   8 – 3i   3 – 2i |
      | -206  -48  -34 -104 |      | 1 + 3i   1 – 2i   4 + 2i   3 +  i |
      |  -39   92   27   30 |      | 2 -  i   1 + 4i  -3 + 4i   1 +  i |
```

that is, to find matrices **`R`** so that **`R*R = A`**. Apart from softly weeping, what can you do about it ? Don’t despair, we’ll see that a few lines of code will do and, for the real case, even an **HP-15C** will suffice for dealing with up to 4x4 matrices.

Squarely computing square roots of square matrices

Computing the square root of a matrix is an infrequent operation which is fraught with difficulties. To begin with, only *square matrices* are eligible, of course, but then, unlike the case of a real or complex scalar value, which always has *exactly two* square roots, a matrix can have *more than two* square roots (for instance, the NxN ***identity matrix*** has at least $\mathbf{2^N}$ square roots) up to *infinity*, and can also have *none* whatsoever, not even considering the whole complex domain for its elements.

Further, algorithms for computing the matrix square root when it does exist, are usually ridden with *numerical instabilities*, often failing to converge even when theoretically possible, or needing a large number of iterations, with the usual problem of rounding errors accumulating over time. That’s the case when trying to use a straight matricial version of Newton’s method to solve the equivalent matricial equation **`R*R = A`**. It *should* converge fast but frequently it does *not*, ultimately resulting in divergence, so more stable methods are definitely required.

Taking these difficulties into account and as the square root of a matrix is rarely seen outside of advanced engineering applications, it’s not surprising that *no* calculator does include this functionality as a built-in feature, even those having extensive matrix capabilities. But should the need arise, what’s one to do ?

Well, the theory is pretty complicated and we won't deal here with the existence and number of square roots of a given matrix nor the diverse methods available for computing them and their numerical properties, you can *"google the web"* for the details if interested. Instead, we'll be satisfied with a couple of pretty simple but workable implementations of a good, numerically stable algorithm (see below) which will (often) find a square root for any given matrix if one does exist: one version for the *general* case (real or *complex* matrix), implemented in **HP-71B**'s **BASIC**, the other for the *real* matrix case, implemented in **RPN** for the **HP-15C**.

1. General case: Program listing for the HP-71B

This 7-line (253-byte) subprogram, **`MATSQRT`**, takes a *real* or *complex* NxN matrix as input and attempts to find one of its square roots. If none can be found it returns the ***zero matrix*** instead. A convergence indicator is shown while running.

```
SUB MATSQRT(A(,)) @ N=UBND(A,1) @ T=TYPE(A)
IF T=5 THEN REAL B(N,N),C(N,N),D(N,N)
IF T=7 THEN COMPLEX B(N,N),C(N,N),D(N,N)
MAT B=IDN @ FOR T=1 TO 20 @ MAT C=INV(B) @ MAT C=A+C @ MAT C=(.5)*C
MAT D=INV(A) @ MAT D=B+D @ MAT D=(.5)*D @ MAT B=A-C @ MAT A=C
N=FNORM(B) @ DISP N; @ IF N/FNORM(A)<.0000001 THEN END
MAT B=D @ NEXT T @ MAT A=ZER
```

Notes:

- You can use whatever line numbering suits you, no line numbers are internally addressed. **Math ROM** matrix keywords are extensively used.
- A test is made to determine the *type* (real/complex) of the input matrix (which is assumed to be square), and 3 similar auxiliary matrices are then created. Thus *complex* operations and matrices are used/returned *if and only if* the argument matrix is *complex*, else everything is kept *real*, saving memory and run time. **`SHORT`** type matrices aren't supported.
- If convergence isn't achieved within **20** iterations, no square root was found and the ***zero matrix*** is returned instead. The calling program can easily check for this by testing the result matrix **`FNORM`** against zero. You can also increase the iteration limit (**`20`**) and try again if desired.
- The *relative tolerance* to terminate is set to **`1E-7`** which should be adequate in most cases to optimize precision vs. computation speed.

Usage instructions

Simply execute the statement:

```
CALL MATSQRT(A)
```

from the command line or a program, where **`A`** is the real or complex square matrix whose square root you want to find. The result will be returned in **`A`** itself if the square root was indeed found, else the ***zero matrix*** will be returned. You can easily check the result by computing **`A*A`**, which should be very close to the original **`A`**.

Example

1. Find a square root of the 4x4 complex matrix:

$$A = \begin{vmatrix} 4+i & 7+i & 3-i & 4+2i \\ 6-i & 9+4i & 8-3i & 3-2i \\ 1+3i & 1-2i & 4+2i & 3+i \\ 2-i & 1+4i & -3+4i & 1+i \end{vmatrix}$$

First of all, let’s create the matrix and input its elements from the keyboard:

```
>DESTROY A @ OPTION BASE 1 @ COMPLEX A(4,4) @ MAT INPUT A [ENTER]

   A(1,1) ?  (4, 1),(7, 1),( 3,-1),(4, 2)  [ENTER]
   A(2,1) ?  (6,-1),(9, 4),( 8,-3),(3,-2)  [ENTER]
   A(3,1) ?  (1, 3),(1,-2),( 4, 2),(3, 1)  [ENTER]
   A(4,1) ?  (2,-1),(1, 4),(-3, 4),(1, 1)  [ENTER]
```

Now let’s compute and display its square root, in `FIX 4`. After each iteration, a *convergence indicator* is displayed, which must converge to **0** if the matrix square root is to be found:

```
>FIX 4 @ CALL MATSQRT(A) @ MAT DISP A;    [ENTER]

    9.7340  7.7617  3.6823  ...  0.0001   3.9004E-9   { convergence achieved ! }

(0.9868,-0.0946)  ( 2.0348,-0.1254)  ( 0.9028, 0.5128)  ( 1.0584, 1.3773)
(1.1578,-0.6776)  ( 2.8900, 1.0990)  ( 0.9221,-0.8419)  (-0.1454,-0.4297)
(0.0655, 1.1255)  (-0.0061,-0.9580)  ( 2.6403, 0.2270)  ( 1.2978, 0.0147)
(1.2080,-0.0028)  (-0.3845, 0.7936)  (-1.2190, 0.4988)  ( 1.1247,-0.5958)
```

So one of the square roots of matrix **A** is :

$$R = \begin{vmatrix} 0.9868-0.0946i & 2.0348-0.1254i & 0.9028+0.5128i & 1.0584+1.3773i \\ 1.1578-0.6776i & 2.8900+1.0990i & 0.9221-0.8419i & -0.1454-0.4297i \\ 0.0655+1.1255i & -0.0061-0.9580i & 2.6403+0.2270i & 1.2978+0.0147i \\ 1.2080-0.0028i & -0.3845+0.7936i & -1.2190+0.4988i & 1.1247-0.5958i \end{vmatrix}$$

Let’s check the result by squaring it, to see if we get the original **A**:

```
>MAT A=A*A @ MAT DISP A;  [ENTER]

(4.0000, 1.0000)  (7.0000, 1.0000)  ( 3.0000,-1.0000)  (4.0000, 2.0000)
(6.0000,-1.0000)  (9.0000, 4.0000)  ( 8.0000,-3.0000)  (3.0000,-2.0000)
(1.0000, 3.0000)  (1.0000,-2.0000)  ( 4.0000, 2.0000)  (3.0000, 1.0000)
(2.0000,-1.0000)  (1.0000, 4.0000)  (-3.0000, 4.0000)  (1.0000, 1.0000)
```

which indeed *does* reproduce the original matrix **A** to the precision shown. A second root would be **-R** and still others might possibly exist.

2. Real case: Program listing for the HP-15C version

This small, 45-step **RPN** routine for the **HP-15C** implements a modified version of the above program that does require *one less auxiliary matrix*, thus allowing the **HP-15C** to compute the real square root of real square matrices *up to 4x4* with room to spare, no mean feat. It is assumed that the input matrix is stored as matrix **`A`**, which will be replaced by one of its square roots. A convergence indicator is shown after each iteration, which should converge to the *dimension* of the matrix.

```
   01  LBL A           16  RCL MATRIX B    31  RCL MATRIX B
   02  RCL DIM A       17  1/X             32   +
   03  STO I           18  RESULT A        33  RESULT B
   04  DIM B           19  RCL MATRIX A    34   2
   05  DIM C           20   +              35   /
   06  CLX             21  RESULT C        36  RESULT C
   07  STO MATRIX B    22  RCL MATRIX C    37  RCL MATRIX A
   08  MATRIX 1        23   -              38   *
   09  X<> 0           24  RESULT A        39  MATRIX 8
   10  LBL 0           25  RCL MATRIX A    40  X^2
   11  STO+ 0          26   2              41  PSE
   12 uSTO B           27   /              42  RND
   13  GTO 0           28  RESULT C        43  RCL I
   14  LBL 1           29  X<>Y            44  TEST 6
   15  RESULT C        30  1/X             45  GTO 1
```

Notes:

- Unlike the **HP-71B** routine above, *there's no limit* on the number of iterations. You can watch convergence by checking the indicator briefly displayed after each iteration, which should converge to the *dimension* of the input matrix (i.e., to **4** for a 4x4 matrix) if a square root can be found
- The program has found a square root and automatically stops if the convergence indicator equals the matrix dimension when rounded to the display setting specified (i.e., **`FIX 4`**, say), so you can control both precision and running time by setting the display mode before running the program.
- If the convergence indicator fails to converge, you should stop the routine manually. This may happen if the input matrix doesn't have square roots.
- Step 12 is a "User" **`STO`** instruction and ***must*** be entered in **`USER`** mode.

Usage instructions

Just reserve space for 3 matrices with **`DIM (i)`**, set your chosen **`FIX`** display mode, have your input matrix stored in memory as matrix **`A`** (up to 4x4), and execute:

`GSB A` (or simply **`A`** in **`USER`** mode)

Once convergence within the specified display setting is achieved, the program stops with the result stored in **`A`** itself. You can check the result by computing **`A*A`**, which should be very close to the original **`A`**.

Examples

1. Find a square root of the 3rd order Hilbert matrix:

$$A = \begin{vmatrix} 1 & 1/2 & 1/3 \\ 1/2 & 1/3 & 1/4 \\ 1/3 & 1/4 & 1/5 \end{vmatrix}$$

Let’s create the matrix and input its elements from the keyboard:

```
0, DIM (i), MATRIX 0, 3, ENTER, DIM A, MATRIX 1, USER,
  1,  STO A,  .5,  STO A, 3, 1/X, STO A,
X<>Y, STO A, X<>Y, STO A,  .25,   STO A,
X<>Y, STO A, X<>Y, STO A,  .2,    STO A, USER, FIX 4
```

Now let’s compute and display its square root, in **FIX 4**. After each iteration, the convergence indicator is displayed, which will converge to **3** (this is a 3x3 matrix) in some two and a half minutes or so:

```
GSB A -> 8,755.4469 -> 573.0348 -> 44.0363
      ->     6.6614 ->   3.3353 ->  3.0105
      ->     3.0000 ->   3.0000      { convergence achieved ! }
```

To output the computed square root simply output **A**’s elements as usual:

```
MATRIX 1, USER, RCL A, RCL A, ..., RCL A, USER
```

and you’ll get:

$$SQRT(A) = \begin{vmatrix} 0.9174 & 0.3455 & 0.1976 \\ 0.3455 & 0.3750 & 0.2709 \\ 0.1976 & 0.2709 & 0.2959 \end{vmatrix}$$

2. Find a square root of the 4x4 matrix:

$$A = \begin{vmatrix} 56 & 97 & 17 & 89 \\ 33 & -68 & -42 & 5 \\ -206 & -48 & -34 & -104 \\ -39 & 92 & 27 & 30 \end{vmatrix}$$

Let’s create the matrix and input its elements from the keyboard (remember to use **CHS** to enter negative values):

```
 0, DIM (i), MATRIX 0, 4, ENTER, DIM A, MATRIX 1, USER,
 56, STO A,  97, STO A,  17, STO A,   89, STO A,
 33, STO A, -68, STO A, -42, STO A,    5, STO A,
-206, STO A, -48, STO A, -34, STO A, -104, STO A,
 -39, STO A,  92, STO A,  27, STO A,   30, STO A, USER, FIX 4
```

Now let’s compute and display its square root, in **FIX 4**. This time convergence will be to **4** (it’s a 4x4 matrix) in about four and a half minutes:

```
GSB A -> 6,040.2650 -> 383.7246 -> 29.7370 -> 6.6446
      ->     4.9362 ->   4.0163 ->  4.0000 -> 4.0000
```

Again, let's output the matrix square root:

```
MATRIX 1, USER, RCL A, RCL A, ..., RCL A, USER
```

and get:

```
             |  8.0000   6.0000   1.0000   7.0000 |
             | -7.0000  -1.0001  -8.0000   3.0000 |
SQRT(A) =    | -8.0001   6.0000   8.0000  -6.0000 |
             |  6.0000   7.0000   7.0000   3.0000 |
```

which is exact to 4 decimal places save a couple of *ulps* here and there.

3. Attempt to find a square root of the 2x2 matrix:

```
    | 0  1 |
A = |      |
    | 0  0 |
```

As always, let's create the matrix and input its elements from the keyboard:

```
0, DIM (i), MATRIX 0, 2, ENTER, DIM A, MATRIX 1, USER,
0, STO A,  1, STO A,  0, STO A, STO A, USER, FIX 4
```

Now let's attempt to compute and display its square root, in `FIX 4`. This time convergence should be to **2** (it's a 2x2 matrix), but ...

```
GSB A -> 9.999999 99 -> 9.999999 99 -> ... press R/S, CF 9
```

... it doesn't converge so we had to stop it manually (and `CF 9` to stop the display blinking due to the overflow condition). This is a *singular matrix* which simply *doesn't have any square roots at all*, even allowing for complex elements. Matter of fact, for singular matrices like this one (Determinant = 0) the existence of square roots isn't guaranteed but depends on the structure of the elementary divisors of the matrix corresponding to the zero eigenvalues.

Algorithms used

Both versions of the program are based on the ***Denman and Beavers*** matrix square root iteration, which is pretty simple, *quadratically* convergent and numerically *stable* (though it may take a while for the convergence to begin in earnest):

$$P_0 = \mathbf{A},\ \ Q_0 = \mathbf{I},$$

$$\{\ P_{k+1} = 1/2*(P_k + Q_k^{-1}),\ Q_{k+1} = 1/2*(Q_k + P_k^{-1})\ \}\quad \text{for } k = 0, 1, 2, ...$$

using different termination criteria specifically taylored for each particular version.