5th August, 2023, 22:24 (This post was last modified: 23rd August, 2023 20:18 by Valentin Albillo.)

Post: #1

Valentin Albillo
Senior Member

Posts: 999
Joined: Feb 2015
Warning Level: 0%

[VA] SRC #015 - HP-15C & clones: Big NxN Matrix Inverse & Determinant

Hi all,

Welcome to this new ***SRC #15*** I've just created to commemorate the recent general availability of the **HP-15C *Collector's Edition***, which boasts *enormously* increased processing speed (up to *180x*) and memory (up to *3x* more *RAM*,) relative to the original.

However, there's a *significant* ***problem*** which needs addressing.

> **Note**: This ***SRC#15*** was actually intended to be a full article, namely ***HP Article VA058 - Boldly Going - HP-15C CE Big Matrix Woes***, but I considered important to adress the problem *ASAP*, so I've posted outright this abridged version of the *PDF* article. The full article features much more information such as step-by-step comments and stack contents, algorithms used and addtional code (input/output, timings for large matrices and more.) **Stay tuned !**

The Problem

The problem is that although the memory is vastly increased in most *15C* clones (physical or virtual), allowing the dimensioning and use of matrices larger than ***8x8***, some very important advanced matrix operations (namely *matrix inversion*, *system solving* and *determinant*) can't be confidently performed using such large matrices (unreliable results, blocked calculator) because of firmware limitations having to do with the internal *LU decomposition*, which are too difficult or even impossible to overcome via *ad-hoc* patches.

The Sleuthing

Fully aware of this, I investigated what could I do about it, and first of all I looked at my own ***NxN Matrix Inversion program for the HP-41C*** which I wrote and published 40 years ago, in order to try and convert it to run on the *CE*, but I sadly realized that the conversion wasn't viable because the *41C* program is *too long (170 steps, 40 registers)* and needs *11* numbered data registers to work, for a grand total of *51* registers. This would severely limit the maximum size of the matrices to process.

Secondly, the *41C* program uses a lot of *flags* (e.g. 13+1 for *13x13* matrices), which the *CE* doesn't have, and additionally it uses a lot of *indirect* register storage and recall using *several* index registers, while the *CE* only has one, register `I`, so this would considerably complicate and slow the conversion, needing special care with the stack contents. This would further increase program size and running time significantly, as the program uses *nested loops* and thus executes many hundreds or even *thousands* of *RPN* instructions to perform the inversion. In short, ***inviable***.

Now, I investigated whether it would be possible to write a program to create and use the *LU decomposition* as the firmware does but without the dreaded *8x8* limitation, and though this would possibly do, it would nevertheless result in long, complicated code which would further need additional registers to work, thus limiting seriously the size of the matrices it could process. It would also need to execute hundreds or thousands of *RPN* instructions, not firmware microcode. ***Viable*** but difficult and with *suboptimal* performance, so

What to do ? *(cue dramatic pause ...)*

My Solution

To adopt an *entirely new* strategy, that's what. Come to think of it, the *real* problem here is that the **HP-15C**'s firmware can't obtain the *LU decomposition* (an so invert/system-solve/compute the determinant) of a matrix larger than *8x8*. Well, let's avoid it altogether by using some strategy which allows us to use the built-in microcoded inversion instruction for speed, but making sure it *never* has to invert a matrix larger than *8x8* ! Enter ***partitioned matrices***.

A *partitioned matrix* is a matrix considered to be partitioned into a number of submatrices, which we'll call ***blocks***. The blocks can be of arbitrary sizes, for example, this *5x5* matrix **M** can be partitioned into four unequal blocks **A** *(4x4)*, **B** *(4x1)*, **C** *(1x4)* and **D** *(1x1)*, like this:

```
      | 3 1 4 1 5 |                 | 3 1 4 1 |       | 5 |
      | 9 2 6 5 3 |                 | 9 2 6 5 |       | 3 |
  M = | 5 8 9 7 9 | = | A B | , A = | 5 8 9 7 | , B = | 9 | , C = | 6 2 6 4 | , D = | 3 |
      | 3 2 3 8 4 |   | C D |       | 3 2 3 8 |       | 4 |
      | 6 2 6 4 3 |
```

and of course the idea is that there are efficient algorithms to deal with such partitioned matrices by interacting directly with their blocks, in particular for *inverting* the matrix or computing its *determinant*, without ever needing to process any block larger than the *largest* one, whose size *we can choose*.

Note that regardless of their sizes, every set of blocks represents the same original matrix and operations performed using them will result in the same outcome, but some sizes are better one way or another. For instance, my implementation here of the algorithm to *invert* a matrix (similarly for computing its *determinant)* requires that it must be partitioned into *4 blocks* and that block **A** must be a *square matrix* no larger than 8x8 (which forces block **D** to also be a square matrix,) because it eventually needs to compute the *inverse* of **A** using the built-in `[1/x]` instruction.

Also very important, to achieve maximum speed it's essential that my routine spends most of its time executing *microcode*, so that's why block **A** must be dimensioned to be as large as possible without exceeding the *8x8* limit and thus for original *NxN* matrices larger than *8x8* we'll dimension **A** to be exactly *8x8*, which forces the sizes of all remaining blocks: **B** will then be *8x(N-8)*, **C** will be *(N-8)x8*, **D** will be *(N-8)x(N-8)* and as long as *N* ≤ ***16*** no block will be larger than *8x8*, as required.

And last but not least, all the code here will run on *any* **HP-15C** version, including the *original*, the ***Limited Edition*** (patched for extra *RAM* or not), the ***Collector's Edition*** (in its default or extended *RAM* modes), the **DM15**'s various versions and firmwares, as well as assorted true emulators.

> **Note**: For *N* < ***9*** no partioning into blocks is necessary. as the **[1/x]** instruction can be used to directly invert the matrix. However, this subroutine can be used if desired, see ***5x5 Toy Example*** below.

The implementation Part 1: Matrix Inversion

This *29-step*, *33-byte* subroutine will invert *in place* an *NxN* partitioned matrix for *N* ≤ ***16*** (subject to available memory). It takes no inputs but the caller (the user or another program) must have previously dimensioned and populated the four blocks with the elements of the original matrix.

Once it returns, the original values in the blocks will have been replaced with those of the computed *inverse matrix*, which the user or the caller program may proceed to output or use as desired. In other words, this subroutine can be called from the keyboard or another program (in which case it could be directly embedded into it if it's called just once) but it doesn't perform any input or output operations, it just does the inversion *in place*, period.

Program listing

```
LBL E           001- 42,21,15
RESULT A        002- 42,26,11
RCL MATRIX A    003- 45,16,11
RCL MATRIX B    004- 45,16,12
RCL MATRIX C    005- 45,16,13
RCL MATRIX A    006- 45,16,11
1/x             007-       15
RESULT E        008- 42,26,15
 x              009-       20
STO MATRIX C    010- 44,16,13
RESULT D        011- 42,26,14
RCL MATRIX B    012- 45,16,12
MATRIX 6        013- 42,16, 6
1/x             014-       15
RESULT E        015- 42,26,15
 x              016-       20
RESULT B        017- 42,26,12
 x              018-       20
CHS             019-       16
RESULT A        020- 42,26,11
RCL MATRIX C    021- 45,16,13
MATRIX 6        022- 42,16, 6
RESULT E        023- 42,26,15
RCL MATRIX D    024- 45,16,14
RCL MATRIX C    025- 45,16,13
 x              026-       20
CHS             027-       16
STO MATRIX C    028- 44,16,13
RTN             029-    43,32
```

Notes:

- This subroutine doesn't use or alter _any_ numbered storage registers, including the three permanent index registers **R0**, **R1** and **RI**.

- It also doesn't use _any_ flags, labels (other than **LBL E**), branching, loops (simple or nested), logic tests of any kind or scalar operations, and it executes *sequentially* from the first step *(001)* to the last step *(029)*, executing each step just _once_, which means it executes ***29*** *user-code instructions* in all, not hundreds or thousands like other approaches, so it runs very fast (e.g. *0.8"* to invert a *9x9* matrix).

- The inversion is performed *in place*: once the process ends the elements of the inverse replace those of the original matrix. *Reinverting* the computed inverse would get back the original matrix.

- If calling it manually from the keyboard, apart from having to dimension four matrices (the blocks) instead of just one (the large original), which takes but a few extra keystrokes, all the remaining keystrokes to input the data, call the subroutine and output the results are *exactly* the _same_ in number, the user's not doing *any* additional work because of the partitioning.

Requirements:

- The *maximum* size *NxN* matrix you can invert depends on the memory available in your physical or virtual device as per this table, which features the **A**, **B**, **C**, **D** partition into blocks that requires the *least* memory (other block sizes would work equally well but would require more registers, as discussed below):

M	A	B	C	D	E	Regs	+Prog	
9x9	8x8	8x1	1x8	1x1	1x8	89	**94**	*{max. size with **CE/96**}*
10x10	8x8	8x2	2x8	2x2	2x8	116	**121**	
11x11	8x8	8x3	3x8	3x3	3x8	145	**150**	
12x12	8x8	8x4	4x8	4x4	4x8	176	**181**	*{ditto with **CE/192**}*
13x13	8x8	8x5	5x8	5x5	5x8	209	**214**	*{ditto with **DM15/M1B**}*
14x14	8x8	8x6	6x8	6x6	6x8	244	**249**	*{no current device}*
15x15	8x8	8x7	7x8	7x7	7x8	281	**286**	*{ditto}*
16x16	8x8	8x8	8x8	8x8	8x8	320	**325**	*{ditto}*

so, e.g. if you want to invert a *9x9* matrix you need to have *94* registers available in the common pool, which includes all 81 elements (distributed among the four blocks) plus the 8 registers automatically allocated for auxiliary matrix **E**, plus 5 registers to hold the *33-byte* subroutine itself, so *81+8+5* = ***94*** registers in all, as seen in the above table. The reason why auxiliary matrix **E** needs to be so big is because the **HP-15C** can't multiply two *8x8* (say) matrices *in place*, a third matrix (**E**) is needed to store the result.

The subroutine would work *as-is*, *unchanged*, for matrices up to and including *16x16*, but as of *August 2023* no physical or virtual devices exist which provide more than *229* registers because the **HP-15C** has a *RAM* limit of *256* registers and *27* of those are used internally, which means that matrices ***14x14*** or larger can't be processed by any currently existing device. However, it might be possible that some future patch or modification could override that restriction, in which case this subroutine could process matrices up to *16x16* without any modifications.

Also, this subroutine can work with a partition into blocks of other sizes (say a *10x10* matrix partitioned in four blocks of size *5x5*) as long as they and the corresponding auxiliary matrix **E** fit in available memory. This can result in speeding up the inversion process, as it's faster to invert two *5x5* blocks than an *8x8* block and a *2x2* block. However, using the recommended partition **A**(*8x8*), **B**(*8x2*), **C**(*2x8*), **D**(*2x2*) requires auxiliary matrix **E** to hold 8x2 = ***16*** elements while using the partition **A**(*5x5*), **B**(*5x5*), **C**(*5x5*), **D**(*5x5*) requires auxiliary matrix **E** to hold 5x5 = ***25*** elements, nine more.

- Block **A** must be *invertible*, i.e. ***det(A) # 0***. Also, ***det(D-CB) # 0***. For most real-life uses this will be the case.

If these conditions aren't met (if in doubt you can check the inverse's correctness by *reinverting* it, which should get the original matrix back, negligible rounding errors aside), you might try *exchanging* or *moving* **rows** and/or **columns** in the original matrix, computing the inverse, and then exchanging or moving *back* the corresponding **columns** and/or **rows** in the computed inverse matrix. You can also try *transposing* the matrix, computing the inverse, and then *transposing back* the inverse.

Worked examples

The following examples will show you how to use the subroutine and further assume that we're using the newly released **HP-15C CE** *(Collector's Edition)* in its default mode, i.e. with just **96** registers available to store matrix elements.

1.- Worked 5x5 Toy Example

Note: The usefulness of this *toy* example (you could do the inversion directly with **1/x**) is *twofold*: first, to get to know how to use the routine and get comfortable using it and second, to ascertain that you've loaded it correctly into program memory by running it and checking the results it produces, without having to tediously and unnecessarily input and output a large number of values.

Invert the following ***5x5*** matrix **M**; the **A**, **B**, **C**, **D** blocks are:

```
      | 3 1 4 1 | 5 |        | 3 1 4 1 |        | 5 |
      | 9 2 6 5 | 3 |        | 9 2 6 5 |        | 3 |
M =   | 5 8 9 7 | 9 | , A =  | 5 8 9 7 | , B =  | 9 | , C = | 6 2 6 4 | , D = | 3 |
      | 3 2 3 8 | 4 |        | 3 2 3 8 |        | 4 |
      | 6 2 6 4 | 3 |
```

- Initialize and dimension the blocks: **A** is *4x4*, **B** is *4x1*, **C** is *1x4* and **D** is *1x1*:

```
CF 8, FIX 4            {ensure disabled complex stack and set 4 decimal places}
1, DIM (i), MATRIX 0   {now MEM: 01 91 05-2}
4, ENTER, DIM A        {now MEM: 01 75 05-2}
1, DIM B               {now MEM: 01 71 05-2}
4, DIM C               {now MEM: 01 67 05-2}
1, ENTER, DIM D        {now MEM: 01 66 05-2}

USER, MATRIX 1         {set USER mode, reset row/col indexes to first element}
```

- Store the 16 elements of block **A**:

```
3, STO A, 1, STO A, 4, STO A, 1, STO A,
9, STO A, 2, STO A, 6, STO A, 5, STO A,
5, STO A, 8, STO A, 9, STO A, 7, STO A,
3, STO A, 2, STO A, 3, STO A, 8, STO A
```

- Store the 4 elements of block **B**:

```
5, STO B, 3, STO B, 9, STO B, 4, STO B
```

- Store the 4 elements of block **C**:

```
6, STO C, 2, STO C, 6, STO C, 4, STO C
```

- Store the single element of block **D**:

```
3, STO D
```

- **Compute the inverse matrix**:

E -> E   1   4   *{nearly instantaneous}*

- Recall the *inverse matrix* elements from blocks **A**, **B**, **C**, **D**:

```
RCL A ->  0.0265, RCL A ->  0.3591, ..., RCL A -> 0.1638   {16 elements}
RCL B -> -0.3685, RCL B -> -0.3254, ..., RCL B -> 0.1054   { 4 elements}
RCL C ->  0.2747, RCL C ->  0.1004, ..., RCL C -> 0.0585   { 4 elements}
RCL D -> -0.3227                                           { 1 element }
```

so we've got the following *inverse matrix* blocks **A'**, **B'**, **C'**, **D'**:

```
     |  0.0265  0.3591  0.0127 -0.0546 |        | -0.3685 |
A' = | -0.2101  0.2124  0.2118 -0.1291 | , B' = | -0.3254 |
     | -0.0408 -0.4286 -0.0612 -0.0408 |        |  0.7347 |
     | -0.0794 -0.0772 -0.0381  0.1638 |        |  0.1054 |

C' = |  0.2747  0.1004  0.0066  0.0585 | , D' = | -0.3227 |
```

and thus the *5x5* inverse matrix **M'** is:

```
     |  0.0265  0.3591  0.0127 -0.0546 -0.3685 |
     | -0.2101  0.2124  0.2118 -0.1291 -0.3254 |
M' = | -0.0408 -0.4286 -0.0612 -0.0408  0.7347 |
     | -0.0794 -0.0772 -0.0381  0.1638  0.1054 |
     |  0.2747  0.1004  0.0066  0.0585 -0.3227 |
```

2.- Worked *9x9* Full-fledged Example

Invert the following ***9x9*** matrix **M**:

```
    | 5  3  4  7  8  0  1  2 | 6 |
    | 6  7  2  0  5  3  4  8 | 1 |
    | 1  0  8  4  2  5  6  7 | 3 |
    | 8  5  0  6  1  4  2  3 | 7 |
M = | 4  2  6  5  3  7  0  1 | 8 |
```

```
| 7  1  3  2  4  8  5  6 | 0 |
| 0  6  1  3  7  2  8  4 | 5 |
| 2  8  7  1  0  6  3  5 | 4 |
| 3  4  5  8  6  1  7  0 | 2 |
```

- The **A**, **B**, **C**, **D** blocks are:

```
    | 5  3  4  7  8  0  1  2 |       | 6 |
    | 6  7  2  0  5  3  4  8 |       | 1 |
    | 1  0  8  4  2  5  6  7 |       | 3 |
A = | 8  5  0  6  1  4  2  3 | , B = | 7 | , C = | 3  4  5  8  6  1  7  0 | , D = | 2 |
    | 4  2  6  5  3  7  0  1 |       | 8 |
    | 7  1  3  2  4  8  5  6 |       | 0 |
    | 0  6  1  3  7  2  8  4 |       | 5 |
    | 2  8  7  1  0  6  3  5 |       | 4 |
```

- Initialize and dimension the blocks: **A** is *8x8*, **B** is *8x1*, **C** is *1x8*, **D** is *1x1*:

```
CF 8, FIX 4               {ensure disabled complex stack and set 4 decimal places}
1, DIM (i), MATRIX 0      {now MEM: 01 91 05-2}
8, ENTER, DIM A           {now MEM: 01 27 05-2}
1, DIM B                  {now MEM: 01 19 05-2}
8, DIM C                  {now MEM: 01 11 05-2}
1, ENTER, DIM D           {now MEM: 01 10 05-2}

USER, MATRIX 1             {set USER mode, reset row/col indexes to first element}
```

- Store the 64 elements of block **A**:

```
5, STO A, 3, STO A, 4, STO A, 7, STO A, 8, STO A, 0, STO A, 1, STO A, 2, STO A,
6, STO A, 7, STO A, 2, STO A, 0, STO A, 5, STO A, 3, STO A, 4, STO A, 8, STO A,
1, STO A, 0, STO A, 8, STO A, 4, STO A, 2, STO A, 5, STO A, 6, STO A, 7, STO A,
8, STO A, 5, STO A, 0, STO A, 6, STO A, 1, STO A, 4, STO A, 2, STO A, 3, STO A,
4, STO A, 2, STO A, 6, STO A, 5, STO A, 3, STO A, 7, STO A, 0, STO A, 1, STO A,
7, STO A, 1, STO A, 3, STO A, 2, STO A, 4, STO A, 8, STO A, 5, STO A, 6, STO A,
0, STO A, 6, STO A, 1, STO A, 3, STO A, 7, STO A, 2, STO A, 8, STO A, 4, STO A,
2, STO A, 8, STO A, 7, STO A, 1, STO A, 0, STO A, 6, STO A, 3, STO A, 5, STO A
```

- Store the 8 elements of block **B**:

```
6, STO B, 1, STO B, 3, STO B, 7, STO B, 8, STO B, 0, STO B, 5, STO B, 4, STO B
```

- Store the 8 elements of block **C**:

```
3, STO C, 4, STO C, 5, STO C, 8, STO C, 6, STO C, 1, STO C, 7, STO C, 0, STO C
```

- Store the single element of block **D**:

```
2, STO D
```

- **Compute the inverse matrix**:

```
E -> E    1  8]   {~ 0.8 sec.}
```

- Recall the inverse matrix elements from blocks **A**, **B**, **C**, **D**:

```
RCL A -> -0.8879, RCL A ->  1.1441, ..., RCL A ->  0.5356  {64 elements}
RCL B ->  0.4953, RCL B -> -0.1670, ..., RCL B -> -0.3966  { 8 elements}
RCL C -> -0.6907, RCL C ->  0.8420, ..., RCL C -> -0.6703  { 8 elements}
RCL D ->  0.2930                                           { 1 element }
```

so we've got the following inverse matrix blocks **A'**, **B'**, **C'**, **D'**:

```
     | -0.8879  1.1441  0.1934 -0.0150  0.9321 -0.7169 -0.2699 -0.8474 |        |  0.4953 |
     |  0.3822 -0.4559 -0.1674  0.0274 -0.4299  0.2991  0.0893  0.4501 |        | -0.1670 |
     | -0.5514  0.7337  0.1725 -0.1023  0.6292 -0.5107 -0.2008 -0.4859 |        |  0.3434 |
A' = |  1.2107 -1.5120 -0.2495  0.1405 -1.2987  0.9816  0.2347  1.0938 | , B' = | -0.5735 |
     |  0.2241 -0.1713 -0.0954 -0.0868 -0.1196  0.1774  0.0830  0.1148 |        | -0.0983 |
     |  0.6076 -0.8621 -0.2155  0.0166 -0.6233  0.6452  0.1842  0.6312 |        | -0.3560 |
     | -0.9092  1.0035  0.2297 -0.0243  0.8488 -0.6830 -0.1314 -0.7941 |        |  0.4877 |
     |  0.6424 -0.6943 -0.0413  0.0732 -0.6939  0.4720  0.1307  0.5356 |        | -0.3966 |

C' = | -0.6907  0.8420  0.2012 -0.0015  0.7832 -0.6369 -0.0921 -0.6703 | , D' = |  0.2930 |
```

and thus the *9x9* inverse matrix **M'** is:

```
      | -0.8879  1.1441  0.1934 -0.0150  0.9321 -0.7169 -0.2699 -0.8474  0.4953 |
      |  0.3822 -0.4559 -0.1674  0.0274 -0.4299  0.2991  0.0893  0.4501 -0.1670 |
      | -0.5514  0.7337  0.1725 -0.1023  0.6292 -0.5107 -0.2008 -0.4859  0.3434 |
      |  1.2107 -1.5120 -0.2495  0.1405 -1.2987  0.9816  0.2347  1.0938 -0.5735 |
 M' = |  0.2241 -0.1713 -0.0954 -0.0868 -0.1196  0.1774  0.0830  0.1148 -0.0983 |
      |  0.6076 -0.8621 -0.2155  0.0166 -0.6233  0.6452  0.1842  0.6312 -0.3560 |
      | -0.9092  1.0035  0.2297 -0.0243  0.8488 -0.6830 -0.1314 -0.7941  0.4877 |
      |  0.6424 -0.6943 -0.0413  0.0732 -0.6939  0.4720  0.1307  0.5356 -0.3966 |
      | -0.6907  0.8420  0.2012 -0.0015  0.7832 -0.6369 -0.0921 -0.6703  0.2930 |
```

Notes:

- If in doubt, a simple way to *check* the inverse's correction is just to *reinvert* it once you've computed it and written down its elements, which should still be undisturbed in memory. Simply run the subroutine again, like this:

  E (in *User* mode) or **GSB E** (out of *User* mode) -> `E 1 8`

  and you'll get the original matrix back (ignoring negligible rounding errors), which you can verify by using **RCL** in *User* mode, as we did in the *Examples* above.

- Once you're done with using the subroutine you might want to free memory by resizing all blocks **A**, **B**, **C**, **D** and the auxiliary matrix **E** to ***0x0*** as well as resetting the row/col indexes and reallocating numbered storage registers **R0-R.9**, like this:

  ```
  MATRIX 0, MATRIX 1, 19, DIM (i)
  ```

The implementation Part 2: Determinant

This *15-step*, *18-byte* subroutine (half the size of the **Matrix Inversion** one and three times as fast) will compute and display the determinant of an *NxN* partitioned matrix for *N* ≤ ***16*** (subject to available memory). It takes no inputs but the caller (the user or another program) must have previously dimensioned and populated the four blocks with the elements of the original matrix.

Once it returns, the computed determinant is left in the *X* stack register. In other words, this subroutine can be called from the keyboard or another program (in which case it could be directly embedded into it if it's called just once) but it doesn't perform any input operations, it just computes and leaves the determinant in the *X* stack register (i.e. the display.)

Program listing

```
LBL D           001- 42,21,14
RESULT D        002- 42,26,14
RCL MATRIX D    003- 45,16,14
MATRIX 9        004- 42,16, 9
RCL MATRIX B    005- 45,16,12
LASTX           006-    43 36
1/x             007-       15
RESULT E        008- 42,26,15
RCL MATRIX C    009- 45,16,13
 x              010-       20
RESULT A        011- 42,26,11
MATRIX 6        012- 42,16, 6
MATRIX 9        013- 42,16, 9
 x              014-       20
RTN             015-    43 32
```

Notes:

All the **Notes** for the **Matrix Inversion** subroutine exactly apply, with the following changes:

- It sequentially executes ***15*** *user-code instructions* in all, not hundreds or thousands like other approaches, so it runs very fast (e.g. ~ *0.2"* to compute the determinant of a *9x9* matrix).

- After execution is complete, the original matrix is left irretrievably altered.

Requirements:

All the **Requirements** for the **Matrix Inversion** subroutine apply exactly, with the following changes:

- Block **D** is the one which must be *invertible*, i.e. ***det(D) # 0***. For most real-life uses this will be the case.

  If this condition isn't met follow the guidelines given for **Matrix Inversion** above.

Worked examples

The following examples will show you how to use the subroutine and further assume that we're using the newly released **HP-15C CE** *(Collector's Edition)* in its default mode, i.e. with just **96** registers available to store matrix elements.

1.- Worked 5x5 Toy Example

The *5x5* matrix used in this example is the same as the one used in the corresponding example for **Matrix Inversion** above, the only difference being that now you're asked to compute its ***determinant*** instead of its inverse, so just *exactly* follow the initialization and input instructions indicated there until you are about to perform the computation part, where you should now do the following:

- **Compute the determinant**:

```
D -> -1,813.0000   {nearly instantaneous}
```

As no *inverse matrix* es computed, its output instructions don't apply here.

2.- Worked *9x9* Full-fledged Example

The *9x9* matrix used in this example is the same as the one used in the corresponding example for **Matrix Inversion** above, the only difference being that now you're asked to compute its ***determinant*** instead of its inverse, so just *exactly* follow the initialization and input instructions indicated there until you are about to perform the computation part, where you should now do the following:

- **Compute the determinant**:

```
D -> -10,278,575.95   {~ 0.2 sec.; exact is -10,278,576}
```

As no *inverse matrix* es computed, its output instructions don't apply here.

Notes:

You might consider applying the same finalization instructions as described in the final **Notes** for **Matrix Inversion** above.

The implementation Part 3: All together now

There are two *million-dollar* questions:

1.- Can both subroutines fit at the same time in the default *CE/96* mode ?

Yes, both are completely *standalone* (neither one requires the other or its results) but the ***Inversion*** routine is *33 bytes* long while the ***Determinant*** routine is *18 bytes* long, so they would seem to occupy *51 bytes* together.

However, if you change the final `RTN` instruction at *step 029* in the former by an `R/S` instruction and delete the initial `LBL D` at *step 001* and the `RTN` at *step 015* in the latter, you can concatenate both and the combination will fit in *49 bytes*, which is ***7*** registers, which is *exactly* all the memory that is available for the code if you want to process a ***9x9*** matrix in the standard *CE/96* mode.

But if using the extended *CE/192* mode, the question is moot as both routines can be present at the same time, unmodified, and there's plenty of room for larger matrices or additional programs.

2.- Is it possible to compute both *Inverse* and *Determinant* without having to reintroduce the original data ?

Yes, it is quite possible by using either one of two simple *tricks*.

Assuming both routines are present in program memory, say you want to compute the *inverse* ***and*** the *determinant* of a large matrix without having to reintroduce the matrix elements. Do as follows:

- Dimension the blocks and store the elements of the original matrix in their respective blocks

- Execute subroutine **E** to compute the *inverse* matrix.

- Recall and write down the elements of the inverse matrix

- Now you have two options to avoid reintroducing the original matrix again, namely ***either***

a) ***Reinvert*** the matrix to get back the original matrix by executing subroutine **E** once again, then execute subroutine **D** to compute its *determinant*. ***or***

b) Execute subroutine **D** to compute the *inverse's determinant*, then (out of *User* mode) press **1/x** and you'll get the *determinant* of the *original* matrix, as `DET(A) = 1 / DET(Inverse of A).`

That's all, hope you enjoyed it and find it useful for your own purposes. Constructive comments welcome.

V.

Edit: corrected an omission; the reported values of the sample 9x9 determinant were missing the minus sign, as both are negative. The routine itself is Ok. Thanks, ***Bert*** *!*

6th August, 2023, 00:59 **Post: #2**

ctrclckws
Member

Posts: 86
Joined: Sep 2018

RE: [VA] SRC #015 - HP-15C & clones: Big NxN Matrix Inverse & Determinant

Not constructive, I know, but WOW!.
To follow along, I would need a refresher on linear algebra, and the definitions of terms associated with it.

6th August, 2023, 01:15 (This post was last modified: 6th August, 2023 01:20 by Namir.) **Post: #3**

Namir
Senior Member

Posts: 905
Joined: Dec 2013

RE: [VA] SRC #015 - HP-15C & clones: Big NxN Matrix Inverse & Determinant

> **ctrclckws Wrote:** (6th August, 2023 00:59)
>
> Not constructive, I know, but WOW!.
> To follow along, I would need a refresher on linear algebra, and the definitions of terms associated with it.

Valentine is a **math brain**! He has done many exceptionally smart works. I had asked for help in programming the Vieta's formula relating polynomial coefficients and their roots. These formulas can get very very complicated. Valentine sent me an unbleivebly compact BASIC code for the HP-71B. It was brilliant! The closest code I found on the internet was written in C++ and had a small error (which I was able to find and correct thanks to Valentine's BASIC code)! Valentine BASIC code blew me away!!!

Namir (a.k.a. "The Math Heretic")

6th August, 2023, 12:44 (This post was last modified: 6th August, 2023 12:45 by EdS2.) **Post: #4**

EdS2
Senior Member

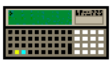
Posts: 538
Joined: Apr 2014

RE: [VA] SRC #015 - HP-15C & clones: Big NxN Matrix Inverse & Determinant

.
An excellent offering, or pair of offerings - thanks Valentin!

As ever, the explanatory material is much appreciated.

6th August, 2023, 16:17 (This post was last modified: 6th August, 2023 19:04 by J-F Garnier.) **Post: #5**



J-F Garnier
Senior Member

Posts: 845
Joined: Dec 2013

RE: [VA] SRC #015 - HP-15C & clones: Big NxN Matrix Inverse & Determinant

Brillant, Valentin, really !

I was expecting something long and complicated, and your solutions are so compact.

As you wrote:

> **Quote:**
>
> - This subroutine doesn't use or alter *any* numbered storage registers, including the three permanent index registers **R0**, **R1** and **RI**.

- It also doesn't use _any_ flags, labels (other than **LBL E**), branching, loops (simple or nested), logic tests of any kind or scalar operations, and it executes *sequentially* from the first step *(001)* to the last step *(029)*, executing each step just _once_, which means it executes ***29*** *user-code instructions* in all, not hundreds or thousands like other approaches, so it runs very fast (e.g. *0.8"* to invert a *9x9* matrix).

Really impressive.
We will need some time to understand the logic behind the code, or better wait for your full VA058 article.

J-F

EMAIL PM WWW FIND QUOTE REPORT

6th August, 2023, 18:27 **Post: #6**

Fernando del Rey
Junior Member

Posts: 24
Joined: Dec 2013

RE: [VA] SRC #015 - HP-15C & clones: Big NxN Matrix Inverse & Determinant

Absolutely brilliant, Valentín!

As said by J-F, we will need some time to understand the matrix partition methods you have used to implement such elegant solutions to both the matrix inversion and determinant problems.

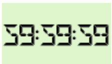
The fact that it can be done with short and linear pieces of code in both cases, with no loops or branching, use of no memory registers or flags, … is indeed hard to believe!

This is also a great example of what can be done with the power of the matrix operations set offered by the 15c. It is worth trying to read and understand your code after having reread the relevant chapter about matrices in the 15c Owner’s Handbook (Section 12).

Thanks for your effort to produce this jewel, Valentín, much appreciated!

EMAIL PM FIND QUOTE REPORT

12th August, 2023, 14:45 **Post: #7**



Werner
Senior Member

Posts: 798
Joined: Dec 2013

RE: [VA] SRC #015 - HP-15C & clones: Big NxN Matrix Inverse & Determinant

I saw this and promptly ordered a 15CE, which I wasn't planning to do ;-)
(I'd request a commission, Valentin)
It arrived 4 days ago, and I have been able to try out Valentin's routine, and get acquainted with the 15C's matrix functions and programming.
I have been able to shave of two lines and 4 bytes of his code, just realizing that the intermediate matrix E could be used throughout, no need for the STO MATRIX instructions:

27 lines, 29 bytes:

```
001 LBL E
002 RCL MATRIX A
003 RCL MATRIX B
004 RCL MATRIX C
005 RCL MATRIX A
006 RESULT A
007 1/x
008 RESULT E
009 x
010 RCL MATRIX B
011 RESULT D
012 MATRIX 6
013 1/x
014 RESULT C
015 x
016 RESULT B
017 x
018 CHS
019 RCL MATRIX E
020 RESULT A
021 MATRIX 6
022 RCL MATRIX D
023 RCL MATRIX E
024 RESULT C
```

025 x
026 CHS
027 RTN

Cheers, Werner

EMAIL PM FIND QUOTE REPORT

13th August, 2023, 03:21 (This post was last modified: 13th August, 2023 03:23 by Valentin Albillo.) **Post: #8**

Valentin Albillo
Senior Member

Posts: 999
Joined: Feb 2015
Warning Level: 0%

RE: [VA] SRC #015 - HP-15C & clones: Big NxN Matrix Inverse & Determinant

.
Hi, **Werner**,

> **Werner Wrote:** (12th August, 2023 14:45)
> I saw this and **promptly ordered a 15CE**, which I wasn't planning to do ;-) (I'd request a commission, Valentin)

Congratulations on a good decision. Knowing you, I'm sure your new **HP-15C CE** is going to inspire you to create amazing code wholesale.

As for the commission, I'm glad to help José G. Divasson sell as many *CEs* as he can, he's done an amazing, *exhausting* work to bring this most awesome calc to all *HP* fans and he deserves all the credit, I'd never take any money from him. Kudos to his charming wife too, who no doubt helped him throughout this venture. And yes, I know you were joking.

> **Quote:**
> I have been able to shave of two lines and 4 bytes of his code, just realizing that the intermediate matrix E could be used throughout, no need for the STO MATRIX instructions: 27 lines, 29 bytes:

Indeed, very well done. And it's a tribute to the **15C**'s outstanding matrix instruction set that this procedure can be implemented at all in so few steps, and to the **CE**'s memory and speed that this code can invert a *9x9* matrix in *0.8"* flat when run.

Thanks for your interest and excellent work and have a nice weekend.
V.

PM WWW FIND EDIT QUOTE REPORT

21st August, 2023, 14:48 **Post: #9**

Bert
Junior Member

Posts: 1
Joined: Aug 2023

RE: [VA] SRC #015 - HP-15C & clones: Big NxN Matrix Inverse & Determinant

Hi Valentin,

Thanks for this patch and great explanation.

You computed the determinant for the second example as follows.

> **Quote:**
> - Compute the determinant:
>
> D -> 10,278,575.95 {~ 0.2 sec.; exact is 10,278,576}

For completeness, you may wish to correct the typo and make the determinant negative.

EMAIL PM FIND QUOTE REPORT

24th August, 2023, 17:02 **Post: #10**

Fernando del Rey
Junior Member

Posts: 24
Joined: Dec 2013

RE: [VA] SRC #015 - HP-15C & clones: Big NxN Matrix Inverse & Determinant

I guess the solutions provided by Valentín for the Large Matrix Inversion and Determinant calculations did not leave much room for improvement, except for the small code optimization provided by Werner.

Perhaps it would be most useful to complete the trilogy with a similar code to solve Large Systems of NxN linear equations, assuming it can be done.

Would you oblige, Valentín, or anyone else?

EMAIL PM FIND QUOTE REPORT

25th August, 2023, 16:43 (This post was last modified: 25th August, 2023 18:49 by Werner.) **Post: #11**

59:59:59 **Werner**
Senior Member

Posts: 798
Joined: Dec 2013

RE: [VA] SRC #015 - HP-15C & clones: Big NxN Matrix Inverse & Determinant

[Update: shaved off two more bytes ...]

> **Fernando del Rey Wrote:** (24th August, 2023 17:02)
>
> Perhaps it would be most useful to complete the trilogy with a similar code to solve Large Systems of NxN linear equations, assuming it can be done.
>
> Would you oblige, Valentín, or anyone else?

Nothing motivates me more than a challenge!
I originally thought it could not be done, because you would need 6 matrices to pull off the same 'partitioning' trick, so I wrote the 'regular' solve.

It could not be done, until I did it ;-)
So here it is, a routine to allow you to solve up to a 13x13 (!) system on the 15C-2 CE.
It uses partitioning of the matrix, much like Valentin's routines.

To solve a system of linear equations, of order n>8

```
M*X=G
```

partition M as before, with block A 8x8:

```
A B    X    E
    *    =
C D    Y    F
```

All we need to do is solve this 2x2 system, overwriting E and F with X and Y.
The problem, of course, is that we don't have the sixth matrix F, so to be able to supply all necessary data,
you have to combine B and E, and D and F:

```
A BE
C DF
```

The routine below will solve this system and put the results in A and C. It uses 1 subroutine level,
and the registers I,0 and 1. The original matrices are all lost, so you can't solve subsequent systems,
but you may of course provide right-hand sides with more than one column.

44 lines, 51 bytes
Enter line 34 in USER mode

```
001-42,21,12  LBL B
002-45,16,13  RCL MATRIX C
003-45,16,12  RCL MATRIX B
004-45,16,11  RCL MATRIX A
005-42,26,12  RESULT B
006-      10  /
007-42,26,14  RESULT D
008-42,16, 6  MATRIX 6
009-   32  1  GSB 1
010-45,16,11  RCL MATRIX A
011-45,16,14  RCL MATRIX D
012-42,26,13  RESULT C
013-      10  /
014-45,16,12  RCL MATRIX B
015-   32  1  GSB 1
016-45,16,13  RCL MATRIX C
017-42,26,11  RESULT A
018-42,16, 6  MATRIX 6
019-   43 32  RTN
020-42,21, 1  LBL 1
021-   44 25  STO I
022-45,23,25  RCL DIM I
023-45,23,13  RCL DIM C
```

```
024-      33  Rv
025-      30  -
026-42,23,11  DIM A
027-42,16, 1  MATRIX 1
028-42,21, 2  LBL 2
029-   45  0  RCL 0
030-   45  1  RCL 1
031-   43 36  LASTX
032-      40  +
033-45,43,24  RCL g (i)
034u   44 11 uSTO A
035-   22  2  GTO 2
036-45,23,25  RCL DIM I
037-   45 25  RCL I
038-42,16, 4  MATRIX 4
039-   43 36  LASTX
040-   43 33  R^
041-42,23,25  DIM I
042-   45 25  RCL I
043-42,16, 4  MATRIX 4
044-   43 32  RTN
```

An example is worth more than 1000 words:

Solve M*X=G, same as Valentin's Toy example, and the right hand side is the first two columns of the identity matrix, so the result is the first two columns of the inverse, for easy comparison:

```
    3 1 4 1 | 5      1 0
    9 2 6 5 | 3      0 1
M = 5 8 9 7 | 9  G = 0 0
    3 2 3 8 | 4      0 0
    --------+--      ---
    6 2 6 4 | 3      0 0
```

So enter

```
        3 1 4 1
A 4x4 = 9 2 6 5
        5 8 9 7
        3 2 3 8

        5 1 0
B 4x3 = 3 0 1
        9 0 0
        4 0 0

C 1x4 = 6 2 6 4

D 1x3 = 3 0 0
```

GSB B

then

```
     0.0265  0.3591
A = -0.2101  0.2124
    -0.0408 -0.4286
    -0.0794 -0.0772

C =  0.2747  0.1004
```

and the solution is then

```
     0.0265  0.3591
    -0.2101  0.2124
X = -0.0408 -0.4286
    -0.0794 -0.0772
     0.2747  0.1004
```

13x13 example
You will need all the memory here, and be in 15C-2 mode.
Do
1 DIM (i) - only registers I,0,1
MATRIX 0 - set all matrix dimensions to 0

Then enter

```
        2 6 1 3 5 1 6 5          5 4 7 6 0 | 1
        0 4 0 4 9 4 2 2          3 8 4 2 6 | 0
        3 8 9 2 1 7 4 9          1 6 4 4 5 | 0
 A 8x8 = 6 8 0 2 7 9 0 4  B 8x6 = 2 3 1 5 0 | 0
        4 5 4 8 5 5 9 5          1 1 8 0 5 | 0
        6 4 7 1 5 8 5 4          0 5 0 2 4 | 0
        5 7 7 3 0 5 6 8          3 7 9 3 9 | 0
        0 9 3 4 2 0 3 4          4 2 9 9 5 | 0

        6 2 3 1 5 2 2 6          0 4 0 1 2 | 0
        5 2 7 5 0 2 0 3          9 5 3 5 4 | 0
 C 5x8 = 3 8 7 2 4 2 6 4  D 5x6 = 4 5 5 2 5 | 0
        2 4 4 5 5 4 6 3          9 0 9 5 8 | 0
        0 4 0 3 2 4 5 5          7 3 1 7 8 | 0
```

(no, I did not enter these manually.
The program used to fill the 13x13 in order ABCD:
first, define B as 8x5 and D as 5x5, that leaves room for the program.

start with 0 STO I, then
 RCL MATRIX A GSB A
 RCL MATRIX B GSB A
 RCL MATRIX C GSB A
 RCL MATRIX D GSB A

(I did not use the built-in RAN# to be able to reproduce these on other calculators)

045 LBL A
046 X<> I
047 MATRIX 1
048 ENTER
049 LBL 3
050 CLX
051 9821
055 x
056 .211327
063 +
064 FRAC
065 E3
067 %
068 INT
069uSTO (i) @ enter in USER mode!
070 GTO 3
071 +
072 FRAC
073 X<> I
074 RTN

Now remove the program lines 45-74, and enlarge B and D by 1 empty column:
 RCL MATRIX B
 MATRIX 4
 6
 ENTER
 8
 DIM B
 RCL MATRIX B
 MATRIX 4

 RCL MATRIX D
 MATRIX 4
 6
 ENTER
 5
 DIM D
 RCL MATRIX D
 MATRIX 4

 Store 1 in the 1,6 element of B:
 1
 ENTER
 ENTER
 6

STO g B

run GSB B, the program takes about a second to produce the results:

```
A
-2.028655-02
-6.059036-02
-1.902323-02
-4.112729-02
-6.215680-03
 1.821154-02
 8.209561-02
 1.687675-02
C
 2.943207-02
 6.494316-02
 4.809902-02
 3.980061-02
-1.146363-01
```

Hope you like it.
Comments, improvements, criticism, all welcome!

Cheers, Werner

27th August, 2023, 00:39 **Post: #12**

Valentin Albillo
Senior Member

Posts: 999
Joined: Feb 2015
Warning Level: 0%

RE: [VA] SRC #015 - HP-15C & clones: Big NxN Matrix Inverse & Determinant

Hi, all,

It seems this thread has run its course, without attracting much attention to be honest but that's life. Thanks for your interest and comments, **Fernando del Rey**, **ctrclckws**, **EdS2**, **Werner** (who managed to shorten my *inversion* code by two steps and further posted a related routine to solve large *systems*,) **J-F Garnier**, **Namir** and last-but-not-least, newcomer **Bert**, who registered just to let me know of a one-character but significant typo, much appreciated.

Now it's time to provide some closure and so I'm posting this *Epilogue* of sorts which includes a short *42-step timing-helper* program which can be used to obtain arbitrarily accurate timings for both my *matrix inverse* and *determinant* routines for large *N*, namely *9 ≤ N ≤ 16*, depending on the particular device's available memory.

Here you'll find the commented ***Program Listing***, the ***Usage Instructions***, the ***Timings*** I've obtained using it on my **HP-15C CE**/*192*, as well as ancillary sections for ***Things To Do*** and ***Miscellanea***.

The program runs on the *CE/192* for *9 ≤ N ≤ 12* (the *CE/96* lacks the necessary memory) as well as any other devices physical or virtual which have enough memory. The **DM15** with firmware *M1B* might be able to also do *N=13*, but no current device has enough memory to do *14 ≤ N ≤ 16* though both this timing program and the routines themselves would work unchanged.

Program listing:

```
001  LBL A        018  RCL MATRIX A   035  LBL 0
002  CF 8         019  GSB 0          036  STO I
003   1           020  RCL MATRIX B   037  MATRIX 1
004  DIM (i)      021  GSB 0          038  LBL 1
005  X<>Y         022  RCL MATRIX C   039   RAN#
006  MATRIX 0     023  GSB 0          040u  STO (i)    (*)
007   8           024  RCL MATRIX D   041  GTO 1
008   -           025  GSB 0          042  RTN
009   8           026   2             043  LBL E         043  LBL D
010  DIM C        027   0             ...  ...       or  ...  ...
011  X<>Y         028  R/S            071  RTN           057  RTN
012  DIM B        029  STO I
013  ENTER        030  LBL 2
014  DIM D        031   GSB E  or  GSB D
015   8           032   DSE I
016  ENTER        033  GTO 2
017  DIM A        034  RTN
```

(*) *Step* *040u* **STO (i)** *must* be entered in **USER** mode.

Notes:

- ***Steps 001-017*** initialize the program (disable the complex stack, allocate all pool registers for matrices, reset all matrices to *0x0*) and dimension all 4 blocks **A**, **B**, **C**, **D** to their respective sizes: *8x8, 8x(N-8), (N-8)x8 and (N-8)x(N-8)*.
- ***Steps 018-028*** fill up all 4 blocks with random values, then stop with the default number of loops *(20)* in the display, ready for the user to start the loops and the timing.
- ***Steps 029-034*** perform the specified number of loops, calling the partitioned *matrix inversion* routine (**LBL E**) or the *determinant* routine (**LBL D**) that many times, then execution ends.
- ***Steps 035-042*** implement a subroutine that fills up with random values the matrix whose descriptor is passed to it in the *X* stack register.
- ***Steps 043-071*** *or* ***043-057*** implement my partitioned *matrix inversion* routine (**LBL E**) or my *determinant* routine (**LBL D**), respectively, whose listings can be found in the first post of this thread.

Usage Instructions:

Note: This step is <u>not</u> necessary, but if you want to check that the program has been correctly entered and works as intended, do this:

`1, STO RAN#, 9, GSB A -> 20`

and if now you recall the contents of the *first* element of matrices **A** and **D** you should get (in **FIX 4**):

`A1,1 = 0.2018, D1,1 = 0.1746`

This program works strictly for *NxN* matrices with *9 ≤ N ≤ 16 (subject to available memory; N ≤ 12 for the CE/192)*. Inputting *N* outside this range will generate an error. To time the *inversion/determinant* of an *NxN* matrix, do as follows:

- First, run the program to dimension and quickly fill up all *NxN* elements with random values, then it stops with the default number of loops *(20)* in the display:

  `FIX 4, N, GSB A -> 20`

  The user can either accept this default (*20* loops i.e. *20 NxN matrix inversions* or *determinants* will be computed) or else key in another number of loops, say *50* or whatever. The more loops, the more time it takes to perform them all but the greater the timing accuracy. Assuming the user can get the final time accurate to the nearest second, the timing accuracy will be *1/#loops* i.e. *1/20 = 0.05"* for the default.

- Now the user must *simultaneously* <u>press</u> **R/S** *and* <u>start</u> the timing, closely watching for the program to end and taking note of the *final time*.

  R/S -> E 1 8 *(say, depends on N)* for the *inversion* or some numeric value for the *determinant*.

  and dividing the *final time* by the *number of loops* will give the *time per NxN operation* accurate to *1/#loops*.

Timings:

These are the times I obtain on my *CE/192* when using the above program with new, fresh lithium batteries (yours may vary slightly.) Note that I use the *default* ***20*** *loops* to time the *inversion* routine but change it to ***50*** *loops* when timing the *determinant* routine, as it is about *3x* faster:

			Inverse		*Determinant*	
Dim	**Block A**	**Block D**	**<u>20</u> loops**	**time**	**<u>50</u> loops**	**time**
9x9	8x8	1x1	18"	**0.90"**	17"	**0.34"**
10x10	8x8	2x2	24"	**1.20"**	22"	**0.44"**
11x11	8x8	3x3	31"	**1.55"**	29"	**0.58"**
<u>12x12</u>	8x8	4x4	40"	**<u>2.00</u>"**	37"	**<u>0.74</u>"**

Thus, when using my *matrix inversion* routine, inverting a *9x9* matrix takes my **HP-15C CE** less than *one second*, and inverting a sizable *12x12* matrix takes about *two seconds* flat.

As for the *determinant* routine, it can compute the determinant of a *9x9* matrix in *1/3* of a second and that of a *12x12* matrix in less than *3/4* of a second. Thus, it can compute *three 9x9 determinants per second*. That's fast !

Things To Do:

For those interested, there's a number of further things to do if you feel like it, e.g.:

2. In my *OP* I provided routines for *matrix inversion* and *determinant* computations but, as **Fernando del Rey** pointed out, I (purposefully) didn't provide one for *system solving*, which would complete the *"matrix trilogy"* and might be quite useful for large systems.

   I may eventually post mine here but in the meantime **Werner** has recently posted a *212-step, 238-byte* system solving program which doesn't use or rely on my code, and just a few days ago posted another version but this time in the spirit of my routines, i.e. dealing with suitably *partitioned matrices*, which is thus *5x* shorter and *10x* faster than his previous standard attempt. This further proves that when solving large systems in the memory-expanded **HP-15C** *partitioning* is the way, which was the whole point of this *SRC#15 !*

   Be as it may, there are other ways to try and solve *systems* and as I said above, I may eventually post one or two different attempts, not doing it right now because documenting and formatting those routines per my standards does take a lot of time. Meanwhile, other people's routines are most welcome.

2. My routine for *matrix inversion* takes a 4-block *partitioned NxN* matrix as input and does the inversion in place, so the *matrix inverse* is also in the same form and can be manually output by the user or utilized by some other program *as-is*. However, it might be useful to write two supporting, complementary routines:

   a. to convert a *non-partioned NxN* matrix to a 4-block *partitioned* matrix, as needed by the *matrix inverse* and *determinant* routines.

   b. to convert the *partitioned NxN* matrix inverse to a single *non-partitioned NxN* matrix. This might simplify further operations with the *matrix inverse* and would also free 3 matrix descriptors for other purposes.

3. Call my inversion routine from one program of yours to perform, say, *Nth-degree Polynomial Fit* or *Nth-degree Least-Squares Polynomial Regression* or *Multivariate Linear Regression*, or any other tasks which would benefit from using an *NxN matrix inverse* or *determinant* for $9 \le N \le 12$ (subject to available memory.)

Miscellanea:

- Once **R/S** is pressed to start the timing of the *matrix inversion* routine, my *CE* immediately displays one or two ***running*** messages in very quick succession, then the display remains *blank* for the whole duration of the loops. This seems *very* similar to what happens with the dreaded *LE*'s **PSE** *(Pause)* bug (except there's *no* **PSE** instruction whatsoever in my program/routines,) so perhaps the cause is somehow related.

  On the other hand, if timing the *determinant* routine, upon pressing **R/S** a single ***running*** message is immediately displayed *once* and it stands *still*, never blinking or blanking till the timing process ends. All in all, I find the display of the ***running*** message to be quite haphazard, sort of yet "unfinished business".

- While checking this timing-helper program, it accidentally tried to invert a *9x9* matrix using the buil-in **1/x** function (which can't reliably work for matrices larger than *8x8*,) and it immediately halted with an *Error* message and a *blinking* display. All my bad, the code I wrote to compute the blocks' dimensions was faulty. Fortunately nothing got corrupted *AFAICT*.

Regards.
V.

PM WWW FIND EDIT QUOTE REPORT

30th August, 2023, 18:53 (This post was last modified: 30th August, 2023 21:28 by J-F Garnier.) **Post: #13**

J-F Garnier
Senior Member

Posts: 845
Joined: Dec 2013

RE: [VA] SRC #015 - HP-15C & clones: Big NxN Matrix Inverse & Determinant

Application to Complex Matrices

The HP-15C is using partitioned matrices to represent a complex matrix for the inversion operation:
```
| X  -Y |
| Y   X |
```
where **X** and **Y** are submatrices holding the real and imaginary parts, respectively (see the 15c Owner's Handbook, p.164 for instance).
The built-in matrix inversion capability of the HP-15C is limited to the *4x4* complex case, since it is using a *8x8* real matrix.

Fortunately the method described by Valentin can also be used to compute the inverse of complex partitioned matrices, thanks to its flexible design:

> **Valentin Albillo Wrote:** (5th August, 2023 22:24)
>
> Also, this subroutine can work with a partition into blocks of other sizes (say a *10x10* matrix partitioned in four blocks of size *5x5*) as long as they and the corresponding auxiliary matrix **E** fit in available memory.

With it, we can now tackle the cases of complex *5x5* and *6x6* matrices on a memory-extended HP-15C.
All we have to do is to set the blocks in line with the complex partitioning scheme, i.e.:

Complex M	A	B	C	D	E	Regs	+Prog
5x5	5x5	5x5	5x5	5x5	5x5	125	**130**
6x6	6x6	6x6	6x6	6x6	6x6	180	**185**

(columns A–E: ---------- Real blocks ----------)

and fill them with **A=X, B=-Y, C=Y, D=X**.

Example of inverting a *5x5* complex matrix :

Invert the following ***5x5*** complex matrix **M**:

```
      | (5,3)  (4,7)  (8,0)  (1,2)  (6,6) |
      | (7,2)  (0,5)  (3,4)  (8,1)  (1,0) |
M =   | (8,4)  (2,5)  (6,7)  (3,8)  (5,0) |
      | (6,1)  (4,2)  (3,7)  (4,2)  (6,5) |
      | (3,7)  (0,1)  (8,7)  (1,3)  (2,4) |
```

- Initialize and dimension the blocks: **A**, **B**, **C** and **D** to *5x5*:

```
1, DIM (i), MATRIX 0
5, ENTER, DIM A
DIM B
DIM C
DIM D
USER, MATRIX 1
```

- Store the real parts of the elements into block **A**:

```
5 STO A, 4 STO A, 8 STO A, 1 STO A, 6 STO A,
7 STO A, 0 STO A, 3 STO A, 8 STO A, 1 STO A,
8 STO A, 2 STO A, 6 STO A, 3 STO A, 5 STO A,
6 STO A, 4 STO A, 3 STO A, 4 STO A, 6 STO A,
3 STO A, 0 STO A, 8 STO A, 1 STO A, 2 STO A
```

- Store the imaginary parts into block **C**:

```
3 STO C, 7 STO C, 0 STO C, 2 STO C, 6 STO C,
2 STO C, 5 STO C, 4 STO C, 1 STO C, 0 STO C,
4 STO C, 5 STO C, 7 STO C, 8 STO C, 0 STO C,
1 STO C, 2 STO C, 7 STO C, 2 STO C, 5 STO C,
7 STO C, 1 STO C, 7 STO C, 3 STO C, 4 STO C
```

- Copy the matrix **A** to **D**, copy the matrix **C** to **B** and change the sign of the matrix **B** elements:

```
RCL MATRIX A
STO MATRIX D
RCL MATRIX C
STO MATRIX B
RCL MATRIX B
CHS
```

- Compute the inverse matrix:

```
GSB E
```

- Recall the inverse matrix elements from blocks **A** and **C**:

Real parts:

```
RCL A:-0.1197, RCL A:-0.0070, RCL A: 0.0635, RCL A: 0.0322, RCL A: 0.0285,
RCL A: 0.0247, RCL A:-0.0206, RCL A: 0.0232, RCL A:-0.0363, RCL A: 0.0638,
RCL A: 0.0277, RCL A:-0.0416, RCL A: 0.0183, RCL A:-0.0348, RCL A: 0.0439,
RCL A: 0.0397, RCL A: 0.0886, RCL A:-0.0975, RCL A: 0.0319, RCL A:-0.0036,
RCL A: 0.0587, RCL A: 0.0388, RCL A:-0.0482, RCL A: 0.0509, RCL A:-0.0549
```

Imaginary parts:

```
RCL C:-0.0115, RCL C:-0.1044, RCL C: 0.0663, RCL C: 0.0940, RCL C:-0.1395,
RCL C:-0.1038, RCL C:-0.0652, RCL C:-0.0403, RCL C: 0.0796, RCL C: 0.0945,
RCL C: 0.0430, RCL C: 0.0005, RCL C:-0.0459, RCL C:-0.0329, RCL C: 0.0005,
RCL C:-0.0081, RCL C: 0.1066, RCL C:-0.0765, RCL C:-0.0271, RCL C: 0.0520,
RCL C:-0.0181, RCL C: 0.0712, RCL C: 0.0676, RCL C:-0.1183, RCL C: 0.0114
```

and thus the *5x5* inverse matrix **M'** is:

```
      | (-0.1197,-0.0115) (-0.0070,-0.1044) (0.0635,0.0663)   (0.0322,0.0940)   (0.0285,-0.1395) |
      | (0.0247,-0.1038)  (-0.0206,-0.0652) (0.0232,-0.0403)  (-0.0363,0.0796)  (0.0638,0.0945)  |
M' =  | (0.0277,0.0430)   (-0.0416,0.0005)  (0.0183,-0.0459)  (-0.0348,-0.0329) (0.0439,0.0005)  |
      | (0.0397,-0.0081)  (0.0886,0.1066)   (-0.0975,-0.0765) (0.0319,-0.0271)  (-0.0036,0.0520) |
      | (0.0587,-0.0181)  (0.0388,0.0712)   (-0.0482,0.0676)  (0.0509,-0.1183)  (-0.0549,0.0114) |
```

J-F

EMAIL PM WWW FIND QUOTE REPORT

1st September, 2023, 17:25 **Post: #14**

59:59:59 **Werner**
Senior Member

Posts: 798
Joined: Dec 2013

RE: [VA] SRC #015 - HP-15C & clones: Big NxN Matrix Inverse & Determinant

The following routines will solve a real system up to 11x11 (GSB B) or a complex system up to 6x6 (GSB C).

Subroutine B will of course be able to solve a 5x5 complex system the usual way, ie by
transforming it into a 10x10 real system, but a 12x12 real system takes too much space.
Making use of the special structure of a complex linear equation, however, it is possible to go up to a 6x6 using routine C.
Solving 12x12 and 13x13 real systems remains possible, with my previous routine (in this thread) that requires a special setup.
In contrast, these new routines do all the partitioning and re-assembling for you.
Both routines solve A*C=B, so input MATRIX A and B, perform either GSB B or GSB C, and the result will be in MATRIX C.
Additionally, the complex routine expects the matrices to be in c-form, so with real and complex parts alternating:

a11r a11c a12r a12c ...
a21r a21c a22r a22c ...

with suffix -'r' being the real part and -'c' the complex part of a number. See the Owner's handbook pg 160 'Calculations with Complex Matrices' for more information on the various formats of complex matrices and how to switch between them.
The result C will also be in that format.
The drawback of this ease of use is the size of 157 bytes - but you may leave off either routine B or C, depending on your needs.

Overview:

routine B: 45 lines, 58 bytes
solves a real system A*C=B of order n>8
If A is nxn and B is nxm, it uses n*(2*n+m-8) registers, eg solving a single 11x11 system needs 11*15=165 registers

routine C: 46 lines, 59 bytes
solves a complex system A*C=B.
If A is nx2n and B is nx2m, it uses n*(4*n+3*m) registers eg a 6x6 system with 1 right hand side would need 6*27=162 registers

subroutines needed for both routines (35 lines, 40 bytes):

Subroutine 1
splits a matrix (in stack register X) in a top and bottom part; the row size of the top part is in stack register Y.
The matrix itself will be redimensioned to hold the top part only; the bottom part will be in MATRIX C.

Subroutine 2
joins MATRIX C (top) with the matrix in stack register X (bottom). The result will be in MATRIX C.

Subroutine 3
moves elements from the matrix in I to MATRIX C. Used by both subroutine 1 and 2.

6x6 complex example

A
(2,5) (6,5) (1,9) (3,5) (5,6) (1,4)
(6,7) (5,1) (0,5) (4,8) (0,5) (4,4)
(9,5) (4,7) (2,7) (2,3) (3,0) (8,5)
(9,6) (2,8) (1,0) (7,9) (4,3) (9,4)
(6,2) (8,0) (0,3) (2,4) (7,5) (9,4)
(0,7) (4,6) (4,0) (5,3) (4,8) (8,4)

f USER to turn USER mode ON
f MATRIX 0
6 ENTER 12 f DIM A
f MATRIX 1

2 STO A
5 STO A
6 STO A
5 STO A
...
8 STO A
4 STO A

6 ENTER 2 f DIM B
1 STO B

GSB C (or just 'C' in USER mode)

RCL C -7.3378-03
RCL C 2.4292-02
..

to get

C
(-7.3378-03, 2.4292-02)
(1.9606-02,-4.7971-02)
(3.3200-02,-3.5196-02)
(-6.0637-03,-4.2733-02)
(7.6352-02,-1.4428-02)
(-8.7514-02, 3.6901-02)

157 bytes, 126 lines:

```
@ solve A*C=B
@ 157 bytes
@ input A and B, result in C=inv(A)*B
@
@ complex solve A*C=B
@ all in c-form
@ up to order 6
@                    matrix:    A       B       C       D       E
001 LBL C               @      -----------------------------------
002 RCL MATRIX A        @       ABc     XYc
003 Py,x
004 RCL DIM A
005 RCL MATRIX B        @       A       X
006 Py,x                @       B       Y
007 GSB 1               @
008 STO MATRIX E        @       A       X       Y               Y
009 RCL DIM A           @       B
010 RCL MATRIX A
011 GSB 1
012 STO MATRIX D        @       A       X       B       B       Y
013 MATRIX 4
014 CHS
015 RCL MATRIX B
016 MATRIX 4
017 GSB 2
018 MATRIX 4            @       A       X      -BX      B       Y
019 RCL MATRIX A
020 STO MATRIX B
021 RESULT C
022 /
023 STO MATRIX A        @       CX      A       CX      B       Y
024 RCL DIM D
025 RCL MATRIX A
026 MATRIX 4
027 GSB 1
028 RCL MATRIX A
029 MATRIX 4            @       C       A       Xt      B       Y
030 RCL MATRIX D
031 RCL MATRIX C
032 MATRIX 4            @                       X
033 RESULT E
034 MATRIX 6            @                                      Y-BX
035 RCL MATRIX D
036 RCL MATRIX A
037 RESULT B
```

```
038 MATRIX 6            @              A-BC
039 RESULT E
040 /                   @                                      Y/A
041 RESULT C
042 MATRIX 6            @                      X-CY
043 RCL MATRIX E
044 GSB 2
045 Cy,x                @                      XYc
046 RTN

@ real solve A*C=B
@                    matrix:    A       B       C       D       E
047 LBL B               @      -----------------------------------
048 8                   @       AB      X
049 RCL MATRIX B        @       CD      Y
050 GSB 1
051 STO MATRIX E        @       AB      X       Y               Y
052 8                   @       CD
053 RCL MATRIX A
054 GSB 1
055 STO MATRIX D        @       AB      X       CD      CD      Y
056 8
057 RCL MATRIX A
058 MATRIX 4
059 GSB 1
060 MATRIX 4
061 RCL MATRIX A
062 MATRIX 4            @       A       X       B       CD      Y
063 RESULT C
064 /
065 RCL MATRIX B
066 RCL MATRIX A
067 RESULT B            @               -1      -1
068 /                   @              A .X    A .B     CD      Y
069 RCL MATRIX C
070 STO MATRIX A        @       B       X       B       CD      Y
071 8
072 RCL MATRIX D
073 MATRIX 4
074 GSB 1               @                                t
075 MATRIX 4            @       B       X       D       C       Y
076 RCL MATRIX A        @ prepare line 089 ;-)
077 RCL MATRIX D
078 MATRIX 4            @                               C
079 RCL MATRIX B
080 RESULT E
081 MATRIX 6            @                                      Y-C.X
082 RCL MATRIX D
083 RCL MATRIX A
084 RESULT C
085 MATRIX 6            @                      D-C.B
086 RESULT E
087 /
088 RESULT B
089 MATRIX 6            @              X-B.Y
090 STO MATRIX C
091 RCL MATRIX E

@ combine C=X and Z=Y to
@ C= | X |
@ | Y |
@ In: X: Z=Y
@ Z is any matrix but C
@ Out: C= | X |
@ | Y |
@

092 LBL 2
093 STO I
094 RCL DIM C
095 X<>Y
096 CHS
097 MATRIX 1
098 STO- 0
```

```
099 X<>Y

100 LBL 0
101 RCL DIM I
102 X<>Y
103 R^
104 -
105 X<>Y
106 DIM C

@ core subroutine
@ copy I -> C

107 LBL 3
108 RCL 0
109 LASTX
110 +
111 RCL 1
112 RCL g (i)
113uSTO C @ enter in USER mode!
114 GTO 3
115 RCL MATRIX C
116 RTN

@ split Z=| X |
@ | Y |
@ In:
@ Y: rX (#rows of X)
@ X: matrix Z=XY
@ Z is anything but C
@ Out:
@ X: MAT C
@ Z=X and C=Y

117 LBL 1
118 STO I
119 MATRIX 1
120 GSB 0
121 RCL DIM I
122 LASTX
123 X<>Y
124 DIM I
125 RCL MATRIX C
126 RTN
```

Hope you like it,
Cheers, Werner

EMAIL PM FIND QUOTE REPORT

5th September, 2023, 13:19 **Post: #15**

J-F Garnier
Senior Member

Posts: 845
Joined: Dec 2013

RE: [VA] SRC #015 - HP-15C & clones: Big NxN Matrix Inverse & Determinant

> **Werner Wrote:** (1st September, 2023 17:25)
> The following routines will solve a real system up to 11x11 (GSB B) or a **complex system up to 6x6** (GSB C).
> ...

Let's see how to solve a complex 6x6 system using Valentin's core code and a few glue commands:

Werner's *6x6* complex example is to solve **M x X = B** defined as:

```
| (2,5) (6,5) (1,9) (3,5) (5,6) (1,4) |   |   |   | (1,0) |
| (6,7) (5,1) (0,5) (4,8) (0,5) (4,4) |   |   |   | (0,0) |
| (9,5) (4,7) (2,7) (2,3) (3,0) (8,5) |   |   |   | (0,0) |
| (9,6) (2,8) (1,0) (7,9) (4,3) (9,4) | x | X | = | (0,0) |
| (6,2) (8,0) (0,3) (2,4) (7,5) (9,4) |   |   |   | (0,0) |
| (0,7) (4,6) (4,0) (5,3) (4,8) (8,4) |   |   |   | (0,0) |
```

- Invert the *6x6* complex matrix **M** (see details above):

Create the *6x6* **A, B, C** and **D** matrices,
fill the **A** and **C** blocks with respectively the real and imaginary parts of the elements of **M**,
set block **B=-A** and block **D=C**,
invert the partitioned matrix (**A, B, C, D**) with **GSB E**.
The inverted matrix **M'** is in **A** (real parts) and **C** (imaginary parts)

- Now, solve the system by doing **X = M' x B** :

Set the RHS vector as a *6x2* matrix **B** (HP "complex form"):
```
6 ENTER 2 DIM B
```
initialize the **B** values with real parts in column 1 and imaginary parts in column 2:
```
0 STO MATRIX B ; clear all elements
MATRIX 1
1 STO B ; set B(1,1)=1
```
compute **A x B** into **E**:
```
RCL MATRIX A
RCL MATRIX B
RESULT E
x
```
Then, initialize the **B** values with *negated imaginary parts in column 1* and *real parts in column 2*:
```
0 STO MATRIX B ; clear all elements
MATRIX 1
0 STO B, 1 STO B ; set B(1,2)=1 [user mode is ON]
```
add **C x B** to **E**:
```
RCL MATRIX C
RCL MATRIX B
CHS
MATRIX 6 ; this is doing MAT E = E - C x B
```

- The resulting complex vector is in "HP complex form" in **E** with the real parts in column 1 and imaginary parts in column 2, that can be read sequentially with **RCL E** in user mode:

```
(-7.3378-03, 2.4292-02)
( 1.9606-02,-4.7971-02)
( 3.3200-02,-3.5196-02)
(-6.0637-03,-4.2733-02)
( 7.6352-02,-1.4428-02)
(-8.7514-02, 3.6901-02)
```

- Now, what are we missing for a complete complex matrix handling?

I would like very much to have a way to compute the *determinant* of a complex matrix in partitioned form.
But I don't see any way to do it easily.

J-F

EMAIL PM WWW FIND QUOTE REPORT

7th September, 2023, 21:15 **Post: #16**

Valentin Albillo
Senior Member

Posts: 999
Joined: Feb 2015
Warning Level: 0%

RE: [VA] SRC #015 - HP-15C & clones: Big NxN Matrix Inverse & Determinant

Hi, **Jean-François**,

> **J-F Garnier Wrote:** (5th September, 2023 13:19)
> Let's see **how to solve a complex 6x6 system using Valentin's core** code and a few glue commands: [...]

Really admirable, solving a *6x6 complex* system of equations with such ease by using my matrix inversion routine plus a few commands executed directly from the keyboard and above all, *lots* of ingenuity. Very well done, **J-F**, thanks for sharing.

> **Quote:**
> I would like very much to have **a way to compute the *determinant* of a complex matrix in partitioned form**. But I don't see any way to do it easily.

Determinants are overrated . Other than being used to check whether a matrix is singular or to check the resultant of two polynomials or to solve a system using Cramer's rule, I don't know of many practical uses let alone uses of determinants of complex matrices, and the ones I've just mentioned are quite inefficient and best tackled using other means. Besides, usually you just need to check if it's *zero* or not, or at most its *absolute value* (condition numbers, etc.)

That said, computing the *absolute value (modulus)* of the determinant of a *complex* matrix is fairly trivial using partitioned *real* matrices, like this:

Let's have an *NxN complex* square matrix **M = A + iB**, where **A**, **B** are *real* square matrices holding the *real/imaginary parts*, respectively, of its elements. If we then construct this familiar 4-block partitioned *real* matrix

```
M' = | A  -B |
     | B   A |
```

then **SQR(DET(M'))** is the *absolute value* of **DET(M)**. Let's see a numeric example taken from here:

We have

```
      | ( 1, 2) ( 2,  3) ( 3,  1) |     DET(M) = (44, -6)
  M = | (-1, 2) ( 2, -1) (-1, -1) | ,
      | ( 0, 3) (-2,  0) ( 2,  2) |     ABS((44,-6)) = 44.4072066223
```

Now

```
                      |  1   2   3 |       | 2   3   1 |
  M = A + iB, A =     | -1   2  -1 | , B = | 2  -1  -1 |
                      |  0  -2   2 |       | 3   0   2 |
```

So

```
                       |  1   2   3  | -2  -3  -1 |
                       | -1   2  -1  | -2   1   1 |     DET(M') = 1972
  M' = | A  -B | =     |  0  -2   2  | -3   0  -2 | ,
       | B   A |       |  2   3   1  |  1   2   3 |     SQR(1972) = 44.4072066223
                       |  2  -1  -1  | -1   2  -1 |
                       |  3   0   2  |  0  -2   2 |
```

which perfectly agrees and it's trivial to code for the **HP-15C**. Not exactly what you wanted but it might be a start for your ingenuity.

Best regards.
V.

PM WWW FIND EDIT QUOTE REPORT

16th September, 2023, 15:24 **Post: #17**

J-F Garnier
Senior Member

Posts: 845
Joined: Dec 2013

RE: [VA] SRC #015 - HP-15C & clones: Big NxN Matrix Inverse & Determinant

> **Valentin Albillo Wrote:** (7th September, 2023 21:15)
>
> > **J-F Garnier Wrote:** (5th September, 2023 13:19)
> >
> > I would like very much to have **a way to compute the *determinant* of a complex matrix in partitioned form**. But I don't see any way to do it easily.
>
> [...] computing the *absolute value (modulus)* of the determinant of a *complex* matrix is fairly trivial using partitioned *real* matrices [...].
>
> Let's have an *NxN complex* square matrix **M = A + iB**, where **A**, **B** are *real* square matrices holding the *real/imaginary parts*, respectively, of its elements. If we then construct this familiar 4-block partitioned *real* matrix
>
> ```
> M' = | A  -B |
>      | B   A |
> ```
>
> then **SQR(DET(M'))** is the *absolute value* of **DET(M)**.

That's really interesting !
Actually, I had a idea in mind when I asked for a way to compute the determinant of a partitioned complex matrix.
The HP-71B Math ROM is internally using the same partitioned scheme to invert complex matrices or solve complex systems, and for the same reason as the 15C, doesn't have a function to compute the determinant of complex matrices.
Clearly, the 71B algorithm originated from the 15C. The algorithm then changed with the 28C/S that directly computes a LU decomposition of the complex matrix, and thus was able to compute the complex value of the determinant.

So it could have been a nice feature of the 71B Math ROM to return the absolute value of the determinant with the DETL function, following a complex matrix inversion operation or system solving. It would have cost almost nothing (computing the determinant from the LU decomposition is immediate), and may have been useful as you noted to estimate the condition number.

Thanks Valentin for all !

J-F

Today, 22:19 **Post: #18**

Valentin Albillo
Senior Member

Posts: 999
Joined: Feb 2015
Warning Level: 0%

RE: [VA] SRC #015 - HP-15C & clones: Big NxN Matrix Inverse & Determinant

.
Hi, **J-F**,

> **J-F Garnier Wrote:** (16th September, 2023 15:24)
> That's really interesting ! [...] **Thanks Valentin for all !**

You're welcome, **J-F**, thanks to you for your continued appreciation and kind comments.

I've noticed that there's still some things pending in the *"To do"* list I mentioned for this thread, so I might post new things here in the foreseeable future to cater for them if no one else does first (or even if they do !)

In the meantime **I'm separately posting *Part 2* of this thread** in a few minutes, stay tuned ! :)

Best regards.
V.