The Philosophy Works®
Lakeville, Minnesota, USA

Hyperspace Algebra Tools

(a.k.a.: "HAT")

"Be hip: find perp." …see page two

http://ftp.setterholm.com/PseudoInverse/Hat.pdf

Save as: "Hat050.pdf" - version 0.50
Supersedes: August 30th 2011 version 0.40; Originated: August 3rd 2011
HAT.exe is intended to run on 'Windows' operating systems.
Data input is via "comma separated values" .csv files
which spreadsheets generate.

September 14th, 2011

Jeff Setterholm
Systems Engineer

Reduce experience to perceived essentials for good.
TM
Expect little from one mind. -or- Suffer scoundrels.
What better place to think?

Solve for **A**, **B**, & **C** in the example algebra problem:

```
                                                    Output#1    Hints:
  Equation#1:  1.00 * A + 0.20 * B + 0.30 * C =  1.10     A =  1.0
  Equation#2:  0.30 * A + 1.00 * B + 0.10 * C =  2.20     B =  2.0
  Equation#3:  0.10 * A + 0.20 * B + 1.00 * C = -0.50     C = -1.0

This problem has:   three equations:         (K=3)
                    with one output:         (L=1)
                  in three unknowns: A,B,& C (N=3)
```

HAT is a software tool for people (who already know Algebra 1) to *begin* solving:

K-equations, with
L-outputs, in
N-unknowns … for K >=N
(i.e. real-world *algebra* problems.).

For arbitrary problems with more than four-equations in four-unknowns, it's *a waste of time* to use pencil & paper to arrive at accurate numerical solutions, whereas computers can do the legwork in the blink of an eye… once the equations are inside the computer in a well-structured way.

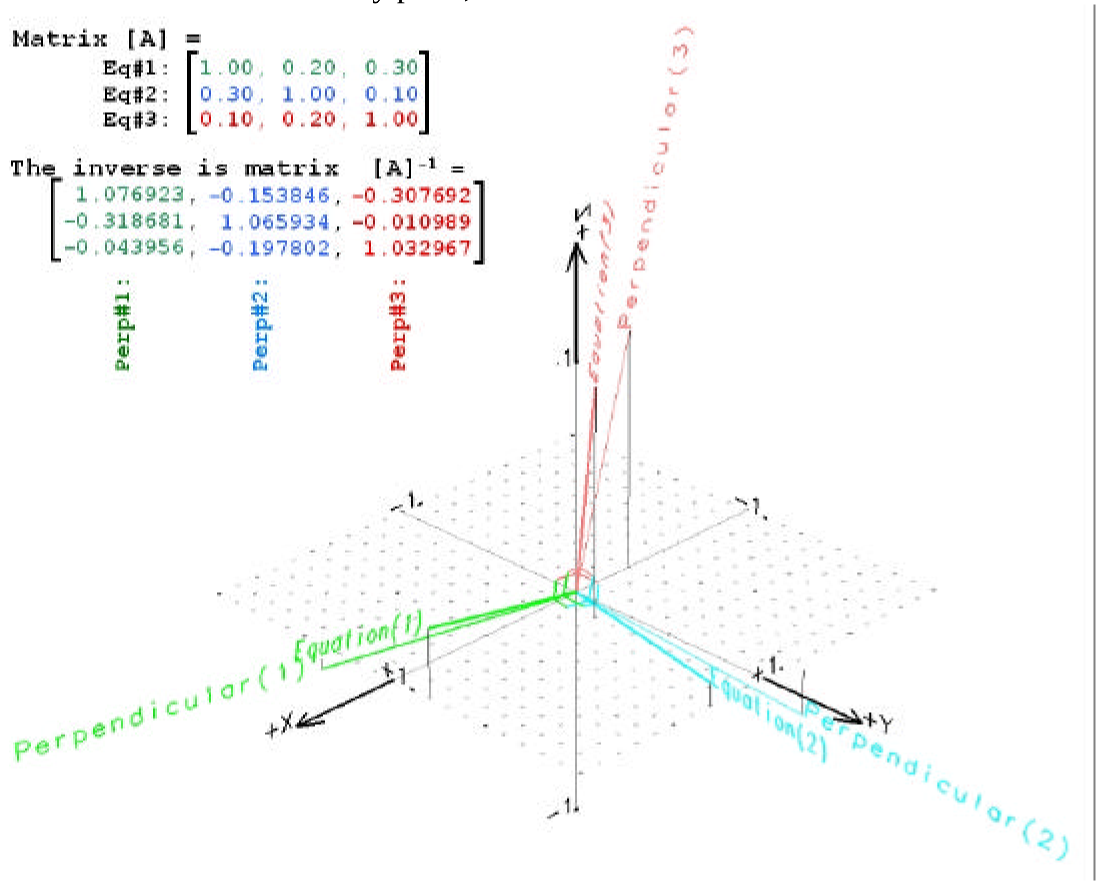
This tool is "a well-structured & automated way"
of having your computer solve
K-equations with L-outputs in N-unknowns.

This tool supports solving problems with many-more-than-three unknowns. Each unknown creates/adds another "dimension" to the "space" in which the problem will be understood and solved; hence having more than three unknowns …more than "a 3-D problem"… **creates a "hyperspace"** (i.e.:>3-D**)**. These algorithms work in hyperspace as well as within the familiar territory of "Algebra 1 land".

This tool is based on the subject: ***linear algebra***; if you liked Algebra 1 *and* were good at it, you may be amazed by the empowerments that linear algebra provides… I'm still amazed, after 35 years of using linear algebra to solve complex applied mathematics problems.

You've probably watched enough science fiction videos to believe that: *hyperspace can be an extremely complicated place*. Even within introductory linear algebra, there are theoretical results whose simple 3-D examples defeat my intuition. HAT provides you with **a carefully chosen, powerful, & relatively simple path** through a complicated forest. So: mastering HAT will leave you far from being "an expert at linear algebra" – but you'll be *analytically empowered* in some marvelous ways.

"Matrix Inversion" is the key piece; here's a look at a matrix inverter's results for the example:



Matrix [A] =

$$\begin{array}{r} \text{Eq\#1:} \\ \text{Eq\#2:} \\ \text{Eq\#3:} \end{array} \begin{bmatrix} 1.00, & 0.20, & 0.30 \\ 0.30, & 1.00, & 0.10 \\ 0.10, & 0.20, & 1.00 \end{bmatrix}$$

The inverse is matrix $[A]^{-1}$ =

$$\begin{bmatrix} 1.076923, & -0.153846, & -0.307692 \\ -0.318681, & 1.065934, & -0.010989 \\ -0.043956, & -0.197802, & 1.032967 \end{bmatrix}$$

The coefficients of the three equations go into the matrix inverter, and three scaled perpendicular directions come out as "answers". Perpendicular to what? … in each case, **perpendicular to *the coefficients of the other two equations***. These scaled directions are entirely independent of what the outputs are equal to; far more powerfully – **these three "perpendiculars" provide all the solutions for all the outputs that the three equations might be equal to**. And matrix inversion works the same way in hyperspace… only human intuition is challenged. Perhaps you're starting to grasp why it might **"be hip to (be able to) find perp."** HAT does that, and more.

The way you solved equations in Algebra 1 took you half way down the road to computing inverses. Here's the output of a "BASIC" program that *does the algebra*; **watch it go:**

```
Example1.Bas 2011.07.13 JMS
          # of equations     # of unknowns     # of outputs
                3                  3                 4
Equations:  Reduce to Identity:       Output#1:  Append an identity matrix:
   1.000000    0.200000    0.300000  :  1.100000   1.000000    0.000000    0.000000
   0.300000    1.000000    0.100000  :  2.200000   0.000000    1.000000    0.000000
   0.100000    0.200000    1.000000  : -0.500000   0.000000    0.000000    1.000000
Row reductions "eliminate" one variable at a time (A, then B, then C):
End of step 1:
>  1.000000    0.200000    0.300000  :  1.100000   1.000000    0.000000    0.000000
   0.000000    0.940000    0.010000  :  1.870000  -0.300000    1.000000    0.000000
   0.000000    0.180000    0.970000  : -0.610000  -0.100000    0.000000    1.000000
End of step 2:
   1.000000    0.000000    0.297872  :  0.702128   1.063830   -0.212766    0.000000
>  0.000000    1.000000    0.010638  :  1.989362  -0.319149    1.063830    0.000000
   0.000000    0.000000    0.968085  : -0.968085  -0.042553   -0.191489    1.000000
End of step 3: Identity matrix:      Answer#1: & Perp (The inverse) plops out:
   1.000000    0.000000    0.000000 A=  1.000000   1.076923   -0.153846   -0.307692
   0.000000    1.000000    0.000000 B=  2.000000  -0.318681    1.065934   -0.010989
>  0.000000    0.000000    1.000000 C= -1.000000  -0.043956   -0.197802    1.032967
Done.                                              ( visualized on page 2 )
```

So, by simply appending an "**identity matrix**" as extra "output" columns (read more about the "identity matrix" on the next page.), **the algebraic solution process yields the full inverse matrix!** Finding perps in hyperspace is quite straightforward… **but the process is tedious, error prone, boring, & inefficient** when *done by hand* (for all but simple problems).

My first three students had difficulty understanding how to replicate the algebra solution above, and my guidance to them was unclear. **Appendix A** (pages 25-33) has the step-by-step ***introductory matrix solver*** details & the solver's BASIC source code; see the details there.

On the preceeding page I claimed that the inverse provides **<u>all</u>** the solutions. As an example: multiplying the inverse matrix times the Output#1 vector yields the Answer#1 vector. As follows:

```
Names  Answer#1                    The inverse                       Output#1
 A= | 1.000000 |      | 1.076923  -0.153846  -0.307692 |     | 1.100000 |
 B= | 2.000000 |    = |-0.318681   1.065934  -0.010989 |  *  | 2.200000 |
 C= |-1.000000 |      |-0.043956  -0.197802   1.032967 |     |-0.500000 |
```

& here's the same multiply *using symbols instead of numbers:*

```
   Vector-out       =              Matrix#1           *   Vector#2
|(j*a + m*b + p*c)|   |    j,          m,          p     |       | a |
|(k*a + n*b + q*c)| = |    k,          n,          q     |  *    | b |
|(l*a + o*b + r*c)|   |    l,          o,          r     |       | c |
```

You can easily check that the numeric & symbolic results agree.

"Identity matrices" (="I" or [I]) generalize **"1.0"** into **hyperspace. I**'s always have exactly as many *rows* as *columns*, with 1.0's along the diagonal and 0.0's everywhere else. To understand more clearly, consider that, in simple algebra: **1.0*X = X**, multiplying by 1.0 doesn't change the value of a number. In the same way, multiplying by **I doesn't alter the hyperspace that you're working in. I** is a richer concept than "1.0"; not only are values preserved, but the inter-dimensional relationships are all preserved as well (up to the number of dimensions that **I** has). To aid your understanding, consider the matrix multiplication [Ai]*[A]: **[Ai]** = the inverse of **[A]** = **[A]**$^{-1}$.

```
      [I]          =               [Ai]             *        [A]
|1.0   0.0   0.0|    |  1.076923 -0.153846 -0.307692|   | 1.00  0.20  0.30|
|0.0   1.0   0.0|  = | -0.318681  1.065934 -0.010989| * | 0.30  1.00  0.10|
|0.0   0.0   1.0|    | -0.043956 -0.197802  1.032967|   | 0.10  0.20  1.00|
```

& the same multiply *using symbols instead of numbers…*

```
                 Matrix-out                  =  Matrix#1  *  Matrix #2
|(j*a+m*b+p*c),(j*d+m*e+p*f),(j*g+m*h+p*i)|     | j, m, p|   | a, d, g|
|(k*a+n*b+q*c),(k*d+n*e+q*f),(k*g+n*h+q*i)|  =  | k, n, q| * | b, e, h|
|(l*a+o*b+r*c),(l*d+o*e+r*f),(l*g+o*h+r*i)|     | l, o, r|   | c, f, i|
```

For any choice of output#'s or answer#'s, **entire (hper)spaces are mapped back to themselves, in both directions**, up to the number of dimensions that the square matrix **I** has. *Division* by [A] is not defined; *multiplying* by [Ai] is as close as you can get, and has much of the same flavor.

For this example problem (but not always true):

```
Output#1=     [A]*Answer#1      &     Answer#1=  [Ai]  *Output#1
Answer#1=[Ai]*[A]*Answer#1      &     Output#1=[A]*[Ai]*Output#1
Answer#1=    [I]  *Answer#1     &     Output#1=  [I]   *Output#1
```

The vector/matrix#1 (on the left) must have exactly as many *columns* as the vector/matrix#2 (on the right) has *rows*; otherwise, the multiplication is undefined. The "answer" vector/matrix has the number of rows of #1 and the number of columns of #2.

```
Let:    Ntot  = number of  rows   of #1
        NMtot = number of columns of #1 = number of  rows  of  #2
        Mtot  =                           number of columns of #2
The "Matrix & vector multiply" code can be written as:
```

```
Dim MatVecOut( Ntot, Mtot) <-This BASIC program does matrix multiplies.
Dim MatVec1(   Ntot,NMtot)        <- …and the values have been put in
Dim MatVec2(  NMtot, Mtot)        <- …and the values have been put in
For N=1 to Ntot
  For M=1 to Mtot
    MatVecOut(N,M)=0.
    For NM=1 to NMtot                      #1          #2
      MatVecOut(N,M)=MatVecOut(N,M)+MatVec1(N,NM)*MatVec2(NM,M)
    Next NM
  Next M
Next N
```

(If you're looking for a computing environment to create your own hyperspace algebra tools, **seek *floating point numbers* with a mimimum of 64-bits**. (~ 12 significant digits). HAT uses 128-bit floats (~24 significant digits) which is called "quad precision" for a 32-bit operating system.)

A computationally minor (*but brilliant & vastly empowering*) step beyond what I've just shown you is the '**matrix pseudoinverse**" - another *nifty twist* on this primrose path through the complicated forest of linear algebra which deals with **having more equations than unknowns**. Having more equations than unknowns is quite common in real-life problems, and provides beneficial opportunities, such as finding a "least-squares best fit" through many data points - which reduces the influence of measurement noise on the solution values of the unknowns. But with more equations than unknowns, **[A]** isn't invertible by itself - because it's *not* a square matrix.
Enter the magic of the **pseudoinverse.**

Appendix B (pages 34-47) has ***the source code for "MUse.bas/.exe, a matrix PseudoInverter, OverWriter, & Linear Dependence eliminator.*** The output if MUse.exe is in "MUseOut.txt"; see the appendix for more details. Systems with *more or less equations than unknowns* are referred to here as **[B:Z]** inputs, which are morphed to **[A:Y]** prior to solving. Finding polynomial coefficients morphs **[B:Z]** to -> **[C:Z]** to -> **[A:Y]**. **[C]** is often a non-square matrix.

Calling the non-square matrix of coefficients **[B]** instead of **[A]**, **the pseudoinverse,** denoted here by **[B]** $^{-P}$ or **[Bp]**

$$[B]^{-P} = ([B]^{T} * [B])^{-1} * [B]^{T}$$

where **[B]**^{T} = the transpose of **[B]=[Bt],** formed by interchanging the rows and columns of **[B]**.

When I said a "computationally minor step", I wasn't kidding.
Forming the transpose is trivial; letting [Bt] = the transpose of [B] = [B]^{T} , in BASIC:

```
Dim B[ Ntot,Mtot)            <-This BASIC program creates the transpose.
Dim Bt[Mtot,Ntot]
For N=1 to Ntot
     For M=1 to Mtot
          Bt(M,N)=B(N,M)
     Next M
Next N
```

The term "pseudoinverse" is somewhat confusing**:** the inversion is actually a regular inversion being done to the square matrix **[B]**^{T}***[B]**. Hence the "**pseudo-**" part is the brilliant *data compression technique* associated with pre-forming **[B]**^{T}* **[B]** and then multiplying by **[B]**^{T} after the inversion. Furthermore, if you just "want the answers" rather than "the space of all the possible answers", the inverting of **[B]**^{T}***[B]** is **not** necessary!

So, while it's true that: **Unknown#1** @ **[B]**$^{-P}$ *** Output#1**
we'll directly solve: (**[B]**^{T}* **[B]**) **:** **[B]**^{T}* * **Output#1]**) instead.
which solves just like: **[A : Y]**

After that we'll look at the numerical values of **[B]**$^{-P}$, which are interesting in their own right if you want to understand what the numbers inside these matrices represent.

Letting: **[A]** = **[B]**^{T}$_{*}$ **[B]**
and: **Y** = **[B]**^{T}$_{*}$ **Output#1**
the system: **[A]:Y** <u>row reduces</u> to: **[I] :~Answer#1** without inversion.
This core **[A]** has the dimensions by the unknowns irrespective of the number of equations.
Likewise, the system: **[A]:Y:[Bt]** row reduces to: **[I] :~Answer#1: [B]** $^{-P}$

Let's go right to a numerical example – five equations in three unknowns.
Adding two equations to the opening example (by exercising: `1.0*x+2.0*y-1.0*z=Output`)

```
        A*      B*      C*       Output#1 (=Z)
Eqn#1:   1.0     0.2     0.3   =  1.1
Eqn#2:   0.3     1.0     0.1   =  2.2
Eqn#3:   0.1     0.2     1.0   = -0.5
Eqn#4:  -1.0     0.3     0.2   = -0.6  <- added
Eqn#5:   0.5    -1.0    -0.3   = -1.2  <- added
```

[A] = **[B]**^{T}$_{*}$ **[B]** **Y** = **[B]**^{T}$_{*}$ **Z**

```
   2.35      -0.28       0.08     :    1.71
  -0.28       2.17       0.72     :    3.34
   0.08       0.72       1.23     :    0.29
```

Simple algebra 1 row reductions solve for Answer#1.

```
Row reductions:
Step  1:
   1.000000  -0.119149   0.034043 :    0.727660
   0.000000   2.136638   0.729532 :    3.543745
   0.000000   0.729532   1.227277 :    0.231787
Step  2:
   1.000000   0.000000   0.074725 :    0.925275
   0.000000   1.000000   0.341439 :    1.658561
   0.000000   0.000000   0.978186 :   -0.978186
Step  3:         [I]                   Answer#1
   1.000000   0.000000   0.000000 :    1.000000 = A
   0.000000   1.000000   0.000000 :    2.000000 = B
   0.000000   0.000000   1.000000 :   -1.000000 = C
Done.
```

Inversion isn't necessary in order to solve more equations than unknowns! Of course, we're not finding all the possible answers, only the *particular* Answer#1.

Where did Answer#1 come from? Understanding *what the numbers in these matrices stand for* may help your intuitive grasp. When you study physics, you'll learn about ***the units*** of numbers and variables, which is the same idea.. For example, if you have an amount of money equal to **2**, you don't know how much money that is *until it has a unit associated with it*, for example **2 dollars**, or perhaps **2 cents**.

The units of the numbers inside the matrices are **rates of change.** Consider the equation of a straight line on a 2-D graph, often written: **Y=m*X+b** where **m** is the *slope* of the line and **b** is the Y- intercept of the line with the graph's Y-axis.

m is the *rate of change* of **Y** with respect to changes in the value of **X**.

In calculus, rates of change of outputs with respect to *a single input* are called "**derivatives**"; derivatives are *the local slopes*; they're "local" because, when lines are curved, the slopes change as the input value moves along the curved line.

The two example problems have three "inputs": **A**, **B**, & **C**, which I've referred to as "unknowns". The **numbers** which multiply **A**, **B**, & **C** are slopes of each output with respect to **A**, **B**, & **C**. Since there is more than one input, there's more than one dimension in which to have a slope, in fact there are three slopes associated with each equation. Calculus calls these slopes "***partial* derivatives**" because slopes vary as the direction of measurement of slope varies. Hence:

The numbers inside [A] are <u>numerical partial derivatives</u>.

In algebra 1 as well, the equation coefficients are "numerical partial derivatives".

Changes of notation will simplify and compact all that follows:
In the spirit of: **Y=m*X+b**,

	Single column case:		Multiple column case:
1.	Unknowns will henceforth be an **X** vector	&	unknowns will be an **[X]** matrix
2.	Outputs will henceforth be a **Y** vector	&,	outputs will be a **[Y]** matrix.

```
So:     X    = Unknowns = Answer#1  ,and:  Y   = Outputs = Output#1
        X(1) = A          =  1.0           Y(1)            =  1.1
        X(2) = B          =  2.0           Y(2)            =  2.2
        X(3) = C          = -1.0           Y(3)            = -0.5
```

```
And the elements of [A] are identified by their location in the
matrix:       [A]= | A(1,1)   A(2,1)  A(3,1) |  A(n,m) where
                   | A(1,2)   A(2,2)  A(3,2) |    n   =[1,2,or 3]
                   | A(1,3)   A(2,3)  A(3,3) |      m =[1,2,or 3]
```

After the changes of notation we can write: **Y=[A]*X**

```
[A]= | 1.000000   0.200000   0.300000 |   &  Output#1 = [A} * Unknown#1
     | 0.300000   1.000000   0.100000 |  now written  Y = [A] *    X
     | 0.100000   0.200000   1.000000 |
```

The **units** of each matrix number are the units that <u>go out</u> *to the left* <u>divided by</u> the units that <u>come in</u> *from the right (~ from above)* **during a matrix multiply**. .. which "has to be" because each number is a slope in a particular direction. Assigning units to our first example is enlightening:

The units of **Y** and **X** are usually suggested by the problem itself. Using the fanciful units:

```
  Y(1)="widgets"      X(1)="person"    "/" = the division symbol
  Y(2)="mistakes"     X(2)="hour"          = "per"
  Y(3)="triumphs"     X(3)="dollar"        …creates derivatives
   Y   =          [A]  *  X
```

```
Then the units of the partial derivatives within [A] become:
[A]= | ( widgets/person) ( widgets/hour) ( widgets/dollar) |
     | (mistakes/person) (mistakes/hour) (mistakes/dollar) |
     | (triumphs/person) (triumphs/hour) (triumphs/dollar) |
```

The units must remain consistent during mathematical operations; consider multiplies:

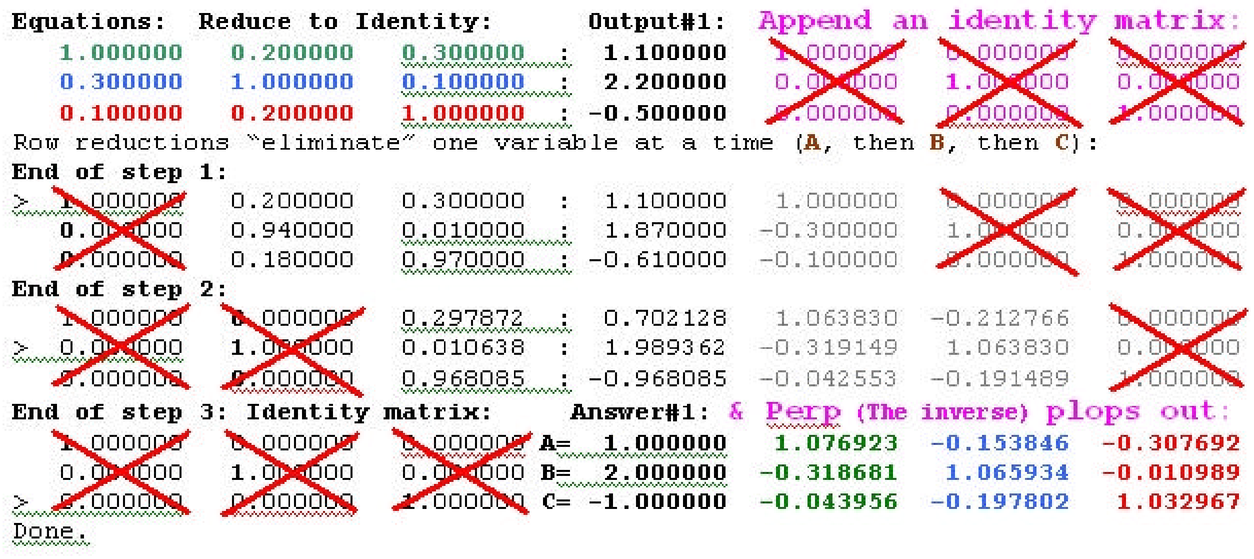
```
Y(1) widgets =   A(1,1) widgets/person * X(1) person
               + A(1,2) widgets/hour   * X(2) hour
               + A(1,3) widgets/dollar * X(3) dollar
```

Both inverse and pseudoinverse matrices have units that are **reciprocal** and **transposed** with respect to the original matrix. That way the resulting units also make sense within multiplies. Units *are consistent* in linear algebra equations. **Units offer an independent way to check equations for correctness.**

The idea of "units" can be abstracted to the unspecified units of the symbols of the variables. So the units of **A(i,j) =** the units of **Y(i)** / the units of **X(j)** and
the units of **B(i,j) =** the units of **Z(i)** / the units of **X(j).**

------ a digression ------

There's a more compact way to compute and display matrix inversions. For the first example, draw X's through the (unnecessary) columns that have *no unexpected information*:



```
Equations:  Reduce to Identity:     Output#1:  Append an identity matrix:
   1.000000   0.200000   0.300000  :  1.100000   1.000000   0.000000   0.000000
   0.300000   1.000000   0.100000  :  2.200000   0.000000   1.000000   0.000000
   0.100000   0.200000   1.000000  : -0.500000   0.000000   0.000000   1.000000
Row reductions "eliminate" one variable at a time (A, then B, then C):
End of step 1:
>  1.000000   0.200000   0.300000  :  1.100000   1.000000   0.000000   0.000000
   0.000000   0.940000   0.010000  :  1.870000  -0.300000   1.000000   0.000000
   0.000000   0.180000   0.970000  : -0.610000  -0.100000   0.000000   1.000000
End of step 2:
   1.000000   0.000000   0.297872  :  0.702128   1.063830  -0.212766   0.000000
>  0.000000   1.000000   0.010638  :  1.989362  -0.319149   1.063830   0.000000
   0.000000   0.000000   0.968085  : -0.968085  -0.042553  -0.191489   1.000000
End of step 3: Identity matrix:     Answer#1: & Perp (The inverse) plops out:
   1.000000   0.000000   0.000000 A=  1.000000   1.076923  -0.153846  -0.307692
   0.000000   1.000000   0.000000 B=  2.000000  -0.318681   1.065934  -0.010989
>  0.000000   0.000000   1.000000 C= -1.000000  -0.043956  -0.197802   1.032967
Done.
```

Hence a matrix can overwrite itself in the course of being inverted; so input:

```
[A]:                                        &  Y:
   1.000000    0.200000    0.300000   :  1.100000
   0.300000    1.000000    0.100000   :  2.200000
   0.100000    0.200000    1.000000   : -0.500000
Goes directly to output:
[A]^-1                                      &  X:
   1.076923   -0.153846   -0.307692   :  1.000000
  -0.318681    1.065934   -0.010989   :  2.000000
  -0.043956   -0.197802    1.032967   : -1.000000
```

HAT's inverter/solver is an overwriter. **Appendix B** has BASIC ***OverWriter source code.***

Let's compute the full pseudoinverse $[B]^{-P}$ of the five-equation example problem using:

$$[B]^{-P}=([B]^T*[B])^{-1}*[B]^T$$

We already have: Transposing [B] and treating it is a [Y] matrix:

$[A]=[B]^T*[B]=$ & $[Y]=[B]^T=$

```
 | 2.35 -0.28  0.08|:| 1.00   0.30   0.10  -1.00   0.50 |
 |-0.28  2.17  0.72|:| 0.20   1.00   0.20   0.30  -1.00 |
 | 0.08  0.72  1.23|:| 0.30   0.10   1.00   0.20  -0.30 |
```

You can use the algorithm that solves the first example problem to find this pseudoinverse; here I'm using the OverWriter because the notation is more compact:

$[Ai]=([B]^T*[B])^{-1}=$ & $[X]=[B]^{-P}=$ **The full pseudoinverse =**

```
| 0.438  0.082 -0.076| : |0.431337   0.205574  -0.016233  -0.428608   0.160012|
| 0.082  0.587 -0.349| : |0.094573   0.576854  -0.223428   0.024503  -0.441566|
|-0.076 -0.349  1.022| : |0.160488  -0.269741   0.944851   0.176135   0.004168|
```

Does **X** @ **$[B]^{-P}$** * **Z**? **<u>Yes</u>.**

```
| 1.000000|   | 0.431337  0.205574 -0.016233 -0.428608  0.160012|   | 1.10|
| 2.000000| = | 0.094573  0.576854 -0.223428  0.024503 -0.441566| * | 2.20|
|-1.000000|   | 0.160488 -0.269741  0.944851  0.176135  0.004168|   |-0.50|
                                                                     |-0.60|
                                                                     |-1.20|
```

And the **units** of $B^{-P}(n,k)$ are: units of **X(n)** /units of **Z(k)**

Does **[I]** = $[B]^{-P}$ * **[B]**? **<u>Yes</u>.**

```
| 1.0  0.0  0.0|  | 0.431337  0.205574 -0.016233 -0.428608  0.160012|  | 1.00   0.20   0.30|
| 0.0  1.0  0.0|= | 0.094573  0.576854 -0.223428  0.024503 -0.441566| *| 0.30   1.00   0.10|
| 0.0  0.0  1.0|  | 0.160488 -0.269741  0.944851  0.176135  0.004168|  | 0.10   0.20   1.00|
                                                                       |-1.00   0.30   0.20|
                                                                       | 0.50  -1.00  -0.30|
```

In this problem: $X=[B]^{-P}*[B]*X=I*X$

And the units of **I(n,m)** are: ~**(** units of **X(n)** / units of **Z(k)** **)** *(units of **Z(k)** / units of **X(n)** **)** *
(…for each k=1 to nEquations…)

which exactly cancel, showing that **[I]** is **unitless**.

And in this example problem: $Z=[B]*[B]^{-P}*Z$, however: $I^1\,[B]*[B]^{-P}$

```
 [B]*[B]^-P=
| 0.498398    0.240022    0.222537   -0.370867    0.072949|
| 0.240022    0.611552   -0.133812   -0.086466   -0.393145|
| 0.222537   -0.133812    0.898542    0.138175   -0.068144|
|-0.370867   -0.086466    0.138175    0.471186   -0.291648|
| 0.072949   -0.393145   -0.068144   -0.291648    0.520321|
```

Here, **Z** is in a five-dimensional space, but *two dimensions of information are lost* in the multiply **$[B]^T$* [B]**, and that information **cannot be recovered** by subsequent multiplies back into a higher dimensional space. **Z ≅[B]*[B]$^{-P}$ *Z** is a least squares best fit of **Z** onto *the 3-D subspace of preserved information*. The example was chosen with **Z** already within the 3-D subspace. In the "real world", the **Z** values are often *experimental measurements* which have at-least tiny **errors**, often called **"noise"**, associated with them. Introducing some reality, let **Z(5)=-1.19** instead of **= -1.20**, and see what happens. In subtle ways, the inputs and outputs are jostled:

```
Now  X=           whereas before X=
    | 1.001600|                 | 1.00|
    | 1.995584|                 | 2.00|
    |-0.999958|                 |-1.00|
```

```
    Z      @    [B]*[B]^-P                                                *   Z
| 1.100729|   | 0.498398    0.240022    0.222537   -0.370867    0.072949|   | 1.10|
| 2.196069|   | 0.240022    0.611552   -0.133812   -0.086466   -0.393145|   | 2.20|
|-0.500681| ≅ | 0.222537   -0.133812    0.898542    0.138175   -0.068144| * |-0.50|
|-0.602916|   |-0.370867   -0.086466    0.138175    0.471186   -0.291648|   |-0.60|
|-1.194797|   | 0.072949   -0.393145   -0.068144   -0.291648    0.520321|   |-1.20|
```

$[A]^{-P}$ ***has remained unchanged***, because the coefficients of the equations, not the particular inputs or outputs, define **$[A]^{-P}$**.

Appendix P, Entry #1, page 49, suggests a useful social purpose which hyperspace mathematics will eventually serve.

So far, the example problems have had **X** as the first power of **A**, **B**, & **C** individually. **Higher-order polynomials offer a much-more-fruitful *generic* approach to finding equations to explain arbitrary data.** HAT least-squares-best-fit's polynomial coefficients to your data. The understanding that the matrix elements are *numerical partial derivatives* is the key to how polynomial fitting works. If you want to determine 12 polynomial coefficients (=unknowns), you'll need to have a minimum of 12 data-points (=equations) to work with.

Often, sensors are calibrated using polynomial fits; people want to know, in advance, how accurate the output of a sensor will be when the sensor outputs emerge from the polynomial that adjusts the raw sensor signals. Having **five times more data-points than the expected number of polynomial coefficients** provides a comfortable margin for finding the actual "best fit". The reason for having more data-points than coefficients is that the solution will be *an exact fit* of the data when #Data-points=#Coefficients – **there is *no* error** - but the resulting polynomial may be *a very inaccurate answer* on either side of the datapoints. Having the coefficients best-fit the larger dataset smoothes out the solution, and also provides a prediction how good the fit is likely to be for another arbitrary real sensor output. Doing polynomial fits on real data *without surplus data & error assessments* is a formula for disaster! The extra data also aids in finding and eliminating the occasional bad data-point, which also helps yield more accurate calibrations.

As an example of how multivariable polynomials are set-up, let's exercise the examples' underlying equation 22 more times to generate a total of "27 datapoints" and then solve for the polynomial coefficients which I'll specify; you'll see that the "**numerical partial derivatives**" of a multivariable polynomial... have "almost-obvious" values once understood.

```
Exercising equation: A *X(1)+ B *X(2)+ C *X(3) = Z
                     1.0*X(1)+2.0*X(2)-1.0*X(3) = Z
```

to synthesize more "datapoints":

#	X(1)	X(2)	X(3):	Z	
1	1.0	0.2	0.3	1.1	<same as before
2	0.3	1.0	0.1	2.2	< "
3	**0.1**	**0.2**	**1.0**	-0.5	< "
4	-1.0	0.3	0.2	-0.6	< "
5	0.5	-1.0	-0.3	**-1.2**	< " (without the added noise)
6	-1.00	0.00	2.00	-3.00	<adding 22 more "datapoints" (#6-#27)
7	-1.00	0.50	-2.00	2.00	
8	-1.00	0.50	0.00	0.00	
9	-1.00	0.50	2.00	-2.00	
10	0.00	-0.50	-2.00	1.00	
11	0.00	-0.50	0.00	-1.00	
12	0.00	-0.50	2.00	-3.00	
13	0.00	0.00	-2.00	2.00	
14	0.00	0.00	0.00	0.00	
15	0.00	0.00	2.00	-2.00	
16	0.00	0.50	-2.00	3.00	
17	0.00	0.50	0.00	1.00	
18	0.00	0.50	2.00	-1.00	
19	1.00	-0.50	-2.00	2.00	
20	1.00	-0.50	0.00	0.00	
21	1.00	-0.50	2.00	-2.00	
22	1.00	0.00	-2.00	3.00	
23	1.00	0.00	0.00	1.00	
24	1.00	0.00	2.00	-1.00	
25	1.00	0.50	-2.00	4.00	
26	1.00	0.50	0.00	2.00	
27	1.00	0.50	2.00	0.00	

The "Order" of a polynomial variable is the highest power of that variable in any particular equation. The coefficient count is one larger than the order, because each variable has a 0th power term as well. Here's a multivariable polynomial that's 2nd order in X(1) and 1st order in X(2) and X(3), so there'll be 12 coefficients – 3x2x2. Using **[C:Z]** as the notation:

```
[C:Z]=
# X1X2X3 X1X2X3 X1X2X3 X1X2X3 X1X2X3 X1X2X3 X1X2X3 X1X2X3 X1X2X3 X1X2X3 X1X2X3 X1X2X3:  Z
   ^0^0^0 ^1^0^0 ^2^0^0 ^0^1^0 ^1^1^0 ^2^1^0 ^0^0^1 ^1^0^1 ^2^0^1 ^0^1^1 ^1^1^1 ^2^1^1
 1  1.     1.     1.     0.2    0.2    0.2    0.3    0.3    0.3    0.06   0.06   0.06   1.1
 2  1.     0.3    0.09   1.     0.3    0.09   0.1    0.03   0.  9  0.1    0.03   0.009  2.2
 3  1.     0.1    0.01   0.2    0.02   0.  2  1.     0.1    0.01   0.2    0.02   0.002 -0.5
 4  1.    -1.     1.     0.3   -0.3    0.3    0.2   -0.2    0.2    0.06  -0.06   0.06  -0.6
 5  1.     0.5    0.25  -1.    -0.5   -0.25  -0.3   -0.15  -0.075  0.3    0.15   0.075 -1.2
 6  1.    -1.     1.     0.     0.     0.     2.    -2.     2.     0.     0.     0.    -3.0
 7  1.    -1.     1.     0.5   -0.5    0.5   -2.     2.    -2.    -1.     1.    -1.     2.0
 8  1.    -1.     1.     0.5   -0.5    0.5    0.     0.     0.     0.     0.     0.     0.0
 9  1.    -1.     1.     0.5   -0.5    0.5    2.    -2.     2.     1.    -1.     1.    -2.0
10  1.     0.     0.    -0.5    0.     0.    -2.     0.     0.     1.     0.     0.     1.0
11  1.     0.     0.    -0.5    0.     0.     0.     0.     0.     0.     0.     0.    -1.0
12  1.     0.     0.    -0.5    0.     0.     2.     0.     0.    -1.     0.     0.    -3.0
13  1.     0.     0.     0.     0.     0.    -2.     0.     0.     0.     0.     0.     2.0
14  1.     0.     0.     0.     0.     0.     0.     0.     0.     0.     0.     0.     0.0
15  1.     0.     0.     0.     0.     0.     2.     0.     0.     0.     0.     0.    -2.0
16  1.     0.     0.     0.5    0.     0.    -2.     0.     0.    -1.     0.     0.     3.0
17  1.     0.     0.     0.5    0.     0.     0.     0.     0.     0.     0.     0.     1.0
18  1.     0.     0.     0.5    0.     0.     2.     0.     0.     1.     0.     0.    -1.0
19  1.     1.     1.    -0.5   -0.5   -0.5   -2.    -2.    -2.     1.     1.     1.     2.0
20  1.     1.     1.    -0.5   -0.5   -0.5    0.     0.     0.     0.     0.     0.     0.0
21  1.     1.     1.    -0.5   -0.5   -0.5    2.     2.     2.    -1.    -1.    -1.    -2.0
22  1.     1.     1.     0.     0.     0.    -2.    -2.    -2.     0.     0.     0.     3.0
23  1.     1.     1.     0.     0.     0.     0.     0.     0.     0.     0.     0.     1.0
24  1.     1.     1.     0.     0.     0.     2.     2.     2.     0.     0.     0.    -1.0
25  1.     1.     1.     0.5    0.5    0.5   -2.    -2.    -2.    -1.    -1.    -1.     4.0
26  1.     1.     1.     0.5    0.5    0.5    0.     0.     0.     0.     0.     0.     2.0
27  1.     1.     1.     0.5    0.5    0.5    2.     2.     2.     1.     1.     1.     0.0
          X(1)^1        X(2)^1               X(3)^1                                     Z
     These are exactly the same as the input data columns.
```

The symbol "^" is used in Basic to indicate "to the power of" i.e. exponentiation; so:

$$X(1)\text{^}\mathbf{2} = X(1)*X(1) = X(1)^2$$

The other columns are likewise ***products of the powers*** of X(1)*X(2)*X(3) **evaluated at each datapoint**. Consider the underlined entry for datapoint #3:

```
          X(1)^2 * X(2)^1 * X(3)^1 = 0.1^2 * 0.2^1 * 1.0^1
                                   =  .01  *  .2   * 1.0
                                   =  .002
```

& **[C:Z]** solves just like **[B:Z]** , (i.e: [A:Y]=[[Ct] * [C] **:** [Ct] * Z] yielding twelve values:

The solved polynomial coefficients are:

```
          PolyCoeff:    Powers:
                    X1^  X2^  X3^
    1        0.0     0    0    0
    2        1.0     1    0    0      =  1.0*X(1)^1
    3        0.0     2    0    0
    4        2.0     0    1    0      =  2.0*X(2)^1
    5        0.0     1    1    0
    6        0.0     2    1    0
    7       -1.0     0    0    1      = -1.0*X(3)^1
    8        0.0     1    0    1
    9        0.0     2    0    1             The other coefficients are 0's.
   10        0.0     0    1    1
   11        0.0     1    1    1
   12        0.0     2    1    1      =  0.0 * X(1)^2 * X(2)^1 * X(3)^1
                     ^    ^    ^   …the powers of each coefficient are added for clarity.
```

Putting the previous noise back in: **Z(5) = - 1.19** & re-solving tweaks all the coefficients.

The polynomial coefficients become:

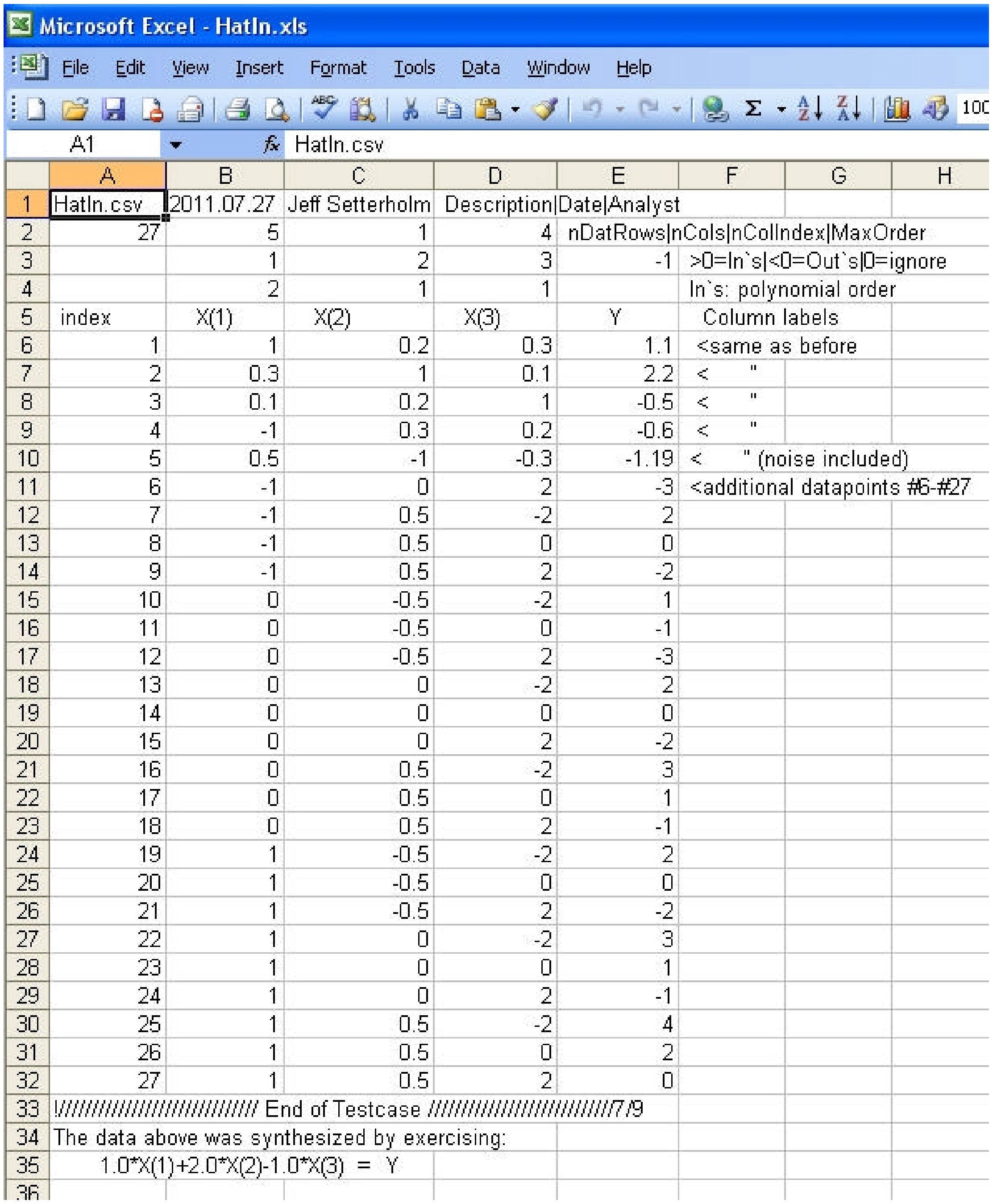
	PolyCoeff:	Powers: X1^	X2^	X3^
1	0.000418	0	0	0
2	**1.00**5937	**1**	**0**	**0**
3	-0.006181	2	0	0
4	**1.99**9261	**0**	**1**	**0**
5	-0.012926	1	1	0
6	0.013015	2	1	0
7	**-1.00**0043	**0**	**0**	**1**
8	-0.003022	1	0	1
9	0.003048	2	0	1
10	0.000214	0	1	1
11	0.006061	1	1	1
12	**-0.00**6171	2	1	1

Prior to putting noise in the data, *there was no error* in this synthesized example. Now we can look at the errors by exercising the resulting polynomial whose coefficients were just computed:

```
Exercising the polynomial:            The errors:
     #        Z:data        Z:poly    Z:poly-Z:data
     1       1.100000      1.100045     0.000045
     2       2.200000      2.198277    -0.001723
     3      -0.500000     -0.499593     0.000407
     4      -0.600000     -0.603655    -0.003655  <- max. error
   Y(5)     -1.190000     -1.193462    -0.003462
     6      -3.000000     -2.999645     0.000355
     7       2.000000      2.000864     0.000864
     8       0.000000      0.000901     0.000901
     9      -2.000000     -1.999061     0.000939
    10       1.000000      1.001087     0.001087
    11      -1.000000     -0.999212     0.000788
    12      -3.000000     -2.999512     0.000488
    13       2.000000      2.000503     0.000503
    14       0.000000      0.000418     0.000418
    15      -2.000000     -1.999667     0.000333
    16       3.000000      2.999920    -0.000080
    17       1.000000      1.000048     0.000048
    18      -1.000000     -0.999823     0.000177
    19       2.000000      2.000636     0.000636
    20       0.000000      0.000499     0.000499
    21      -2.000000     -1.999637     0.000363
    22       3.000000      3.000207     0.000207
    23       1.000000      1.000174     0.000174
    24      -1.000000     -0.999859     0.000141
    25       4.000000      3.999777    -0.000223
    26       2.000000      1.999849    -0.000151
    27       0.000000     -0.000080    -0.000080
```

Appendix C page 48: **Hat.exe** – presently computes (only) polynomial-based solutions.

Here's how the **input data** for the example above looks inside a spreadsheet:


Microsoft Excel - HatIn.xls
File Edit View Insert Format Tools Data Window Help

A1 | HatIn.csv

	A	B	C	D	E	F	G	H
1	HatIn.csv	2011.07.27	Jeff Setterholm	Description|Date|Analyst				
2	27	5	1	4	nDatRows|nCols|nColIndex|MaxOrder			
3		1	2	3	-1	>0=In`s|<0=Out`s|0=ignore		
4		2	1	1		In`s: polynomial order		
5	index	X(1)	X(2)	X(3)	Y	Column labels		
6	1	1	0.2	0.3	1.1	<same as before		
7	2	0.3	1	0.1	2.2	< "		
8	3	0.1	0.2	1	-0.5	< "		
9	4	-1	0.3	0.2	-0.6	< "		
10	5	0.5	-1	-0.3	-1.19	< " (noise included)		
11	6	-1	0	2	-3	<additional datapoints #6-#27		
12	7	-1	0.5	-2	2			
13	8	-1	0.5	0	0			
14	9	-1	0.5	2	-2			
15	10	0	-0.5	-2	1			
16	11	0	-0.5	0	-1			
17	12	0	-0.5	2	-3			
18	13	0	0	-2	2			
19	14	0	0	0	0			
20	15	0	0	2	-2			
21	16	0	0.5	-2	3			
22	17	0	0.5	0	1			
23	18	0	0.5	2	-1			
24	19	1	-0.5	-2	2			
25	20	1	-0.5	0	0			
26	21	1	-0.5	2	-2			
27	22	1	0	-2	3			
28	23	1	0	0	1			
29	24	1	0	2	-1			
30	25	1	0.5	-2	4			
31	26	1	0.5	0	2			
32	27	1	0.5	2	0			
33	!//////////////////////////////// End of Testcase ////////////////////////////////7/9							
34	The data above was synthesized by exercising:							
35	1.0*X(1)+2.0*X(2)-1.0*X(3) = Y							
36								

Export this file in a "**comma separated value**" (**.csv**) format as "HatIn.csv" for use by HAT.

The result is awkward to read:

```
~,2011.07.27,Jeff Setterholm, Description|Date|Analyst
27,5,1,4, nDatRows|nCols|nColIndex|MaxOrder
       ,1,2,3,-1, >0=In`s|<0=Out`s|0=ignore
       ,2,1,1,        , In`s: polynomial order
 index ,    X(1),    X(2),    X(3),       Z,   Column labels
1,1,0.2,0.3,1.1,  <same as before
2,0.3,1,0.1,2.2,"  <      """
3,0.1,0.2,1,-0.5,"  <      """
4,-1,0.3,0.2,-0.6,"  <      """
5,0.5,-1,-0.3,-1.19," <      "" (noise included)"
6,-1,0,2,-3, <additional datapoints #6-#27
7,-1,0.5,-2,2,
8,-1,0.5,0,0,
9,-1,0.5,2,-2,
10,0,-0.5,-2,1,
11,0,-0.5,0,-1,
12,0,-0.5,2,-3
13,0,0,-2,2
14,0,0,0,0
15,0,0,2,-2
16,0,0.5,-2,3
17,0,0.5,0,1
18,0,0.5,2,-1
19,1,-0.5,-2,2
20,1,-0.5,0,0
21,1,-0.5,2,-2
22,1,0,-2,3
23,1,0,0,1
24,1,0,2,-1
25,1,0.5,-2,4
26,1,0.5,0,2
27,1,0.5,2,0
!////////////////////////////// End of Testcase ///////////////////////////7/9
The data above was synthesized by exercising:
       1.0*X(1)+2.0*X(2)-1.0*X(3)  =  Y
```

HAT can be used *<u>without</u> using a spreadsheet* to generate the “HatIn.csv” file. For example:

```
HatIn.csv,2011.07.27,Jeff Setterholm, Description|Date|Analyst
     27,       5,       1,       4, nDatRows|nCols|nColIndex|MaxOrder
       ,      1 ,      2 ,      3 ,      -1 , >0=In`s|<0=Out`s|0=ignore
       ,      2 ,      1 ,      1 ,         , In`s: polynomial order
 index ,    X(1),    X(2),    X(3),       Z,   Column labels
      1,     1.0,     0.2,     0.3,     1.1,  <same as before
      2,     0.3,     1.0,     0.1,     2.2,  <      "
      3,     0.1,     0.2,     1.0,    -0.5,  <      "
      4,    -1.0,     0.3,     0.2,    -0.6,  <      "
      5,     0.5,    -1.0,    -0.3,   -1.19,  <      " (noise included)
      6,   -1.00,    0.00,    2.00,   -3.00, <additional datapoints #6-#27
      7,   -1.00,    0.50,   -2.00,    2.00,
      8,   -1.00,    0.50,    0.00,    0.00,
      9,   -1.00,    0.50,    2.00,   -2.00,
     10,    0.00,   -0.50,   -2.00,    1.00,
     11,    0.00,   -0.50,    0.00,   -1.00,
     12,    0.00,   -0.50,    2.00,   -3.00,
     13,    0.00,    0.00,   -2.00,    2.00,
     14,    0.00,    0.00,    0.00,    0.00,
     15,    0.00,    0.00,    2.00,   -2.00,
     16,    0.00,    0.50,   -2.00,    3.00,
     17,    0.00,    0.50,    0.00,    1.00,
     18,    0.00,    0.50,    2.00,   -1.00,
     19,    1.00,   -0.50,   -2.00,    2.00,
     20,    1.00,   -0.50,    0.00,    0.00,
     21,    1.00,   -0.50,    2.00,   -2.00,
     22,    1.00,    0.00,   -2.00,    3.00,
     23,    1.00,    0.00,    0.00,    1.00,
     24,    1.00,    0.00,    2.00,   -1.00,
     25,    1.00,    0.50,   -2.00,    4.00,
     26,    1.00,    0.50,    0.00,    2.00,
     27,    1.00,    0.50,    2.00,    0.00,
!////////////////////////////// End of Testcase ///////////////////////////7/9
The data above was synthesized by exercising:
       1.0*X(1)+2.0*X(2)-1.0*X(3)  =  Z
```

Hat exports “HatOut.csv” with column formatting resembling the column formatting of the input data. So taking the time to align the columns of your “HatIn.csv” may improve your subsequent documentation and communication of the results achieved using HAT.

Comments about the setup of “HatIn.csv”: each field must be = 34 characters wide.

Line -4:

```
HatIn.csv,  2011.07.27,Jeff Setterholm,      Three commas must follow the three fields.
Description|    Date   |     Analyst                         Remarks are optional.
```

Line -3:

```
     27,        5,        1,        4 ,         Four commas must follow the four fields.
   nDatRows|   nCols |nColIndex|MaxOrder                     Remarks are optional.
```

“MaxOrder” is the highest combined power

In version 0.40 this value must be >0, or the program stops.

Set nColIndex = 0 if you have no index column.

Line -2:

```
        ,       1 ,       2 ,       3 ,      -1 , >0=In`s|<0=Out`s|0=ignore
```

nCols commas must follow the first **nCols** fields.

Remarks are optional.

The columns are reordered, per your assignments above, in “HatOut.csv”, which can then be renamed “HatIn.csv for subsequent editing & use as input.

Line -1:

```
        ,       2 ,       1 ,       1 ,         , In`s: polynomial order
```

nCols commas must follow the first **nCols** fields.

Remarks are optional.

```
        ,         ,         ,         ,         , In`s: polynomial order     is valid
```

Line 0:

```
 index ,     X(1),     X(2),     X(3),       Z,    Column labels
```

nCols commas must follow the first **nCols** fields.

Remarks are optional.

If you don’t provide an” Index” column for your data – HAT will add the column to “HatOut.csv”.

The index is used in accuracy/error reporting.

Line 1: etc

```
      1,      1.0,      0.2,      0.3,     1.1,  <same as before
```

nCols commas must follow the first **nCols** fields. Indices don’t need to sequential or ordered . Remarks are optional.

…

Line 27: HAT expects to read **nDatRows** of data.

```
     27,      1.00,     0.50,     2.00,  0.00,
```

As an example, changing lines -4 to 0 of the example on the previous page to:

```
HatIn.csv,2011.07.27,Jeff Setterholm, Description|Date|Analyst
     5,       5,       1,       1, nDatRows|nCols|nColIndex|MaxOrder
      ,       3 ,      2 ,      1 ,     -1 , >0=In`s|<0=Out`s|0=ignore
      ,       1 ,      1 ,      1 ,        , In`s: polynomial order
index ,    X(1),    X(2),    X(3),       Y,   Column labels
```

Produces “HatOut.csv”:

```
HatOut.csv,2011.07.27,Jeff Setterholm, Description|Date|Analyst
       5,        5,        1,        1, nDatRows|nCols|nColIndex|MaxOrd
        ,        1 ,       2 ,       3 ,      -1 , >0=In`s|<0=Out`s|0=ignore
        ,        1 ,       1 ,       1 ,         , In`s: polynomial order
  Index ,     X(3),     X(2),     X(1),        Z,   Column labels   <- Columns are reordered.
       1,      0.3,      0.2,      1.0,      1.1,  <same as before
       2,      0.1,      1.0,      0.3,      2.2,  <      "
       3,      1.0,      0.2,      0.1,     -0.5,  <      "
       4,      0.2,      0.3,     -1.0,     -0.6,  <      "          <- Data is truncated.
       5,     -0.3,     -1.0,      0.5,    -1.19,  <      "  (noise included)
```

```
The polynomial coefficients are:
       Output#  1,   Powers:
    1,  0.347122655325D-02,   0,   0,   0,   <- the added 0th order term.
    2, -0.100348474693D+01,   1,   0,   0,   <- coefficient of X(3)^1
    3,  0.199547695393D+01,   0,   1,   0,   <- coefficient of X(2)^1
    4,  0.100037796722D+01,   0,   0,   1,   !- coefficient of X(1)^1
```

Data Scaling

Even using 64-bit precision real numbers with ~ 24 significant digits, input data that naturally arises in professional use of HAT (to do polynomial fits) will occasionally produce intermediate computations concurrently both so large and so small that information is lost due to round off error in combining the very large and very small numbers.

Earlier, the equation of a straight line: `Y=`**`m`**`*X+`**`b`** was mentioned wherein **m** is the *slope* and **b** is the *Y-intercept*. Hat scales polynomial data by using m's and b's to adjust data values for better computational advantage and also for better understanding. The names of **m** and **b** are changed to the way that engineers talk - to **Gain** (=m) and **Bias** (=b), Gain and Bias – "**GB**" is my abbrev. - *operate on data columns*; **Gain** *multiplies* the columns data entries (e.g.: vertically expands the data when plotted on a 2-D graph) and **Bias** *shifts* the column entries (e.g. vertically moves the data up or down on a graph without otherwise morphing the data).

Data Scaling – **Pass 1** – *Uniform Bounding - GB1*

GB'ing every data column of "HatIn.csv" to exactly fit the interval [-1.0, 1.0] yields the fact that, no matter how high the order of a polynomial becomes, the numerical partial derivatives will reach but-not-exceed plus-or-minus 1.0. Hence the 24 significant digits will be used to better effect by operating on numbers "that are in the same ballpark". So we seek the GB values for:

```
                    ColumnOut = Gain*ColumnIn + Bias
```

Sort through ColumnIn to find the smallest and largest values: Column**InMin** and Column**InMax**.
We want Column**OutMin** = -1.0 and Column**OutMax** = +1.0. So:

```
Gain = (ColumnOutMax - ColumnOutMin) / (ColumnInMax – ColumnInMin)
     =              2.0             / (ColumnInMax – ColumnInMin)

        When (ColumnInMax–ColumnInMin)=0. (a constant column), set Gain = 1.0

Bias =  ColumnOutMax - Gain * ColumnInMax
     =       1.0     - Gain * ColumnInMax

             When (ColumnInMax–ColumnInMin)=0. this Bias value produces
            ColumnOut=1.0, which seems to have benign downstream effects.
```

GB'ing into the interval [-1.0, 1.0] **often yields polynomial coefficients that are also in the same range**; so when polynomial coefficients are significantly outside the range, *the coefficients may be competing with each other* to force an un-natural fit. (I'm not sure… but watch for large GB1-scaled polynomial coefficients and form your own opinion(s) about what's happening.)

Un-scaling, or **de-scaling** the resulting coefficients to their requested values *isn't easy to do*, but HAT provides. Use of the binomial theorem and Pascal's triangle in a multi-variable, arbitrary-order polynomial environment accomplishes the task, here referred to as "**de-GB'ing**'. If you decide to tackle the algorithms, you may find that **G'ing** & **de-G'ing** are fairly easy to do, whereas **de-B'ing** is rather complicated. I surmise that **Bias shifts** alter (~screw up) the inter-column geometry in hyperspace, whereas **a Gain change** primarily affects the data column involved.

Pass 1 scaling – **GB1** – for the data on page 15 yields:

```
Scaling the input columns into [-1.0,+1.0]
using G1 and B1... as in Y=G1*X+B1:
index     X(1)       X(2)       X(3)   :    Z
G1:     1.0000     1.0000     0.5000     0.2857
B1:     0.0000     0.0000     0.0000    -0.1429
       ---------  ---------  ---------  ---------
   1    1.000000   0.200000   0.150000   0.171429
   2    0.300000   1.000000   0.050000   0.485714
   3    0.100000   0.200000   0.500000  -0.285714
   4   -1.000000   0.300000   0.100000  -0.314286
   5    0.500000  -1.000000  -0.150000  -0.482857
   6   -1.000000   0.000000   1.000000  -1.000000
   7   -1.000000   0.500000  -1.000000   0.428571
   8   -1.000000   0.500000   0.000000  -0.142857
   9   -1.000000   0.500000   1.000000  -0.714286
  10    0.000000  -0.500000  -1.000000   0.142857
  11    0.000000  -0.500000   0.000000  -0.428571
  12    0.000000  -0.500000   1.000000  -1.000000
  13    0.000000   0.000000  -1.000000   0.428571
  14    0.000000   0.000000   0.000000  -0.142857
  15    0.000000   0.000000   1.000000  -0.714286
  16    0.000000   0.500000  -1.000000   0.714286
  17    0.000000   0.500000   0.000000   0.142857
  18    0.000000   0.500000   1.000000  -0.428571
  19    1.000000  -0.500000  -1.000000   0.428571
  20    1.000000  -0.500000   0.000000  -0.142857
  21    1.000000  -0.500000   1.000000  -0.714286
  22    1.000000   0.000000  -1.000000   0.714286
  23    1.000000   0.000000   0.000000   0.142857
  24    1.000000   0.000000   1.000000  -0.428571
  25    1.000000   0.500000  -1.000000   1.000000
  26    1.000000   0.500000   0.000000   0.428571
  27    1.000000   0.500000   1.000000  -0.142857
```

With no further scaling, after [**B]** is expanded to 13 columns to accommodate the 12 polynomial coefficients, the upper left corner of **[A]** =[Ct]*[C] becomes:

```
  27.000000   5.900000  15.350000   2.200000  …
   5.900000  15.350000   5.153000  -1.780000  …
  15.350000   5.153000  15.070700   1.842000  …
   2.200000  -1.780000   1.842000   5.920000  …
     …           …          …          …
```

Further insight can be gained by going through *a second round* of pure-gain adjustment, as you'll see shortly…

Intuition in hyperspace:

Two aspects of hyperspace seem intuitive *to me*:

1. A number called "**the determinant**" of a matrix is **the (signed: ±) hypervolume** enclosed by the vectors. (The outputs columns, if any, aren't part of the determinant). This is just like "area" in 2-D and/or "volume" in 3-D. (The "shapes" are parallelepipeds.)

–and-

2. The "**vector dot product**" between any two columns of the matrix. When each column has a total length of 1.0, the dot product is the cosine of the angle between the vectors. Hence **the angle between vectors can be computed in N-dimensional space** and means the same thing as in 2-D or 3-D. Details about determinants & dot products follow.

1.) It's *difficult* to clearly communicate *the closed-form mathematical expression* for the value of **the determinant**, but *the value "falls out" of the solution processes that we've been using,* with *or without* full matrix inversion. **Starting with the value 1.0, multiply by the values which are used (in division) to reduce the initial matrix to an identity matrix;** presto: the signed hypervolume of the input matrix – ***the determinant*** – materializes; who'd have thought the computation would be that simple? **If that volume goes to zero** – meaning that the input vectors are (somehow) collapsed on themselves, **you might be "dead-in-the-water" using ordinary Algebra 1 techniques to solve a problem;** you have an incomplete set of numerical partial derivatives. Fortunately HAT's matrix inverter has features which bypass determinant=0. hang-ups, a key feature in ease-of-use of the software; the inverter will automatically reduce the size of the system appropriately and give you the next best answer – more on *how this is done* later. *Without a second round of pure-gain adjustment*, the determinant of **[A] "**falls out" as:

```
Determinant (= the signed hypervolume) for the 12-coefficient case:
Row: Column: Fractional contrib:
                  1. *
   1       1     27.000000000000
   7       7     15.206666666667
   2       2     13.946062947538
   8       8      8.163783102623
   3       3      6.039931495913
   4       4      5.218495100420
   9       9      3.402570904799
  10      10      2.166841481094
   5       5      1.101593852033
  11      11      0.731652394579
   6       6      0.418126228482
  12      12      0.153616077359
```
`Determinant= 562370.940460001886`

… which looks like "just another very big, not particularly insightful, number". The **dot product** facilitates *pure-gain adjustment* , so let's consider the dot product.

2.) It turns out that **each individual output element in any matrix multiply** (ref.: page 4) **is a dot product** of the corresponding row vector –and- column vector on the right side of the equation. If **X** and **Y** are ***any two vectors with the same number of elements***, then:

X•Y = "X dot Y"
= a real number (called a "scalar", which is to say – a single number, ≅ "not a vector")
= (X(1)*Y(1) + X(2)*Y(2) + X(3)*Y(3) + X(4)*Y(4) +…etc.)
= (Magnitude of **X**) * (Magnitude of **Y**) * Cosine(of the angle between **X** and **Y** in hyperspace)

---------- ~ End of "Intuition in Hyperspace" ----------

Data Scaling – **Pass 2** – *Pure-Gain Adjustment* – *G2*

To use the dot product for pure-gain adjustment, take the dot product of each data column *with itself*. The angle between a vector and itself is zero; so the cosine of the angle is 1.0. The dot product of column **Y** with itself becomes:

Y•Y = (Magnitude of **Y**) * (Magnitude of **Y**) * 1.0
= (Magnitude of **Y**) ^{2} **hence the square root of this dot product is *the length of* Y.**

So vectors are *pure-gain adjusted to length one* by dividing by the square root of the dot-product of the vector with itself. The idea of *length* also remains intuitive in hyperspace.

After **[B]** is expanded to 13 columns to accommodate the 12 polynomial coefficients, the **G2** pure-gain adjustments for the 13 columns of **[C]** are:

```
G2: 5.196152   3.917908   3.882100   2.433105   1.649364   1.565441      Z
    3.912480   3.006801   3.005653   1.592733   1.227905   1.226062   2.767392
```

Dividing each column of **[C]** by its G2 adjustment, the upper left corner of **[Ct*C]** becomes:

```
   1.000000   0.289812   0.760956   0.174012  …
   0.289812   1.000000   0.338797  -0.186726  …
   0.760956   0.338797   1.000000   0.195012  …
   0.174012  -0.186726   0.195012   1.000000  …
    …          …          …          …        …
```

The values above **are the cosines of the actual angles** between the various columns of data;
the corresponding **actual angles** (in degrees) are:

```
   0.000     73.153     40.451     79.979   …
  73.153      0.000     70.196    100.762   …
  40.451     70.196      0.000     78.755   …
  79.979    100.762     78.755      0.000   …
    …          …          …          …      …
Determinant for the 12-coefficient case:
Row: Column: Fractional contribution:
                  1.0 *
   5       5      1.000000000000
   9       9      0.999983396784
  10      10      0.955558140299
   1       1      0.950096960741
  11      11      0.819076953471
   6       6      0.799783117546
   8       8      0.544194470370
   4       4      0.463868970282
   7       7      0.407156481554
   2       2      0.405694266581
  12      12      0.207186023014
   3       3      0.061769378211
Determinant=      0.000317364340
```

which tells you that only **.031%** of the maximum possible volume (=1.0) is enclosed by the 12 vectors. This gives you a sense of **"how far down toward the noise"** the inverter is going in computing your answers. In contrast, *the four coefficient case* using the same 27 datasets has a much more robust determinant:

```
Determinant for the four-coefficient case:
Row: Column: Fractional contribution:
                  1.0 *
   4       4      1.000000000000
   3       3      0.998569858442
   1       1      0.964149455586
   2       2      0.851506117380
Determinant=      0.819805043087  ~82% of the maximum volume is spanned
```

Each step of the matrix inversion process adds a dimension to the solution. The **fractional contribution** reveals how far out of the accumulating solution hyper-subspace the next dimension protrudes; when the value is less than 1.0, part of that dimension has been consumed by the solution subspace. When the fractional contributions to the determinant = 0.0, the inverter has reached the-end-of-the-line… a collapsed subspace… all the rest of the dimensions are "linearly dependent"… and some dimension(s) of **[A]** will need to be systematically discarded.

So the **G2 pure-gain adjustment** *provides intuitive insight* into what's happening inside the inversion hyper-subspaces, and *reduces computational round off errors* at the same time.

Reviving Collapsed Solutions = "**Eliminating linear dependence(s)**" - a simple example.

Let's go back the opening problem and change Equation#3:

```
         Equation#3 = +2.1*Equation#1 -3.2*Equation#2
```

In a nutshell that's "**linear dependence**": when one vector equals the sum of any combination of the other vectors... which only happens when a vector lies within the hyper-subspace already created by one-or-more other vectors.

Recall that the **fractional contributions** show how far each new vector "**sticks out**" from the previous hyper-subspace; if the new vector doesn't "stick out" at all, then it's linearly dependent… and "dead wood"/useless… *in terms of* aiding the inversion process; the inverter is trying to map the output (hyper)space back into the input (hyper)space, but the inverter can't map back those dimensions of the input (hyper)space wherein *the numerical partial derivatives* are undefined. Proceeding:

```
  Equation#1:  1.00 *A +0.20 *B +0.30 *C=  1.10   *(+2.1)
  Equation#2:  0.30 *A +1.00 *B +0.10 *C=  2.20   *(-3.2)
  Equation#3:  0.10 *A +0.20 *B +1.00 *C= -0.50
```

Revising equation#3 to be *linearly dependent*:

```
  Equation#3:  1.14 *A -2.78 *B +0.31 *C= -4.73
```

Now watch the inverter/solver crunch on this: Appendix B's *OverWriter* solves this in detail on pages 34 thru 37 .

```
Equations:  Reduce to Identity:      Output#1:  Append an identity matrix:
       -1          -2          -3
-1  1.000000   0.200000   0.300000 :  1.100000   1.000000   0.000000   0.000000
-2  0.300000   1.000000   0.100000    2.200000   0.000000   1.000000   0.000000
-3  1.140000  -2.780000   0.310000   -4.730000   0.000000   0.000000   1.000000
```

Row reductions "eliminate" one variable at a time using the largest remaining coefficient first:

```
       -1           2          -3
-1  1.082014   0.000000   0.322302    0.759712   1.000000   0.000000   0.071942
 3 -0.410072   1.000000  -0.111511    1.701439   0.000000   0.000000  -0.359712
-2  0.710072   0.000000   0.211511    0.498561   0.000000   1.000000   0.359712
```

...after the 2nd row reduction:

```
        1           2          -3
-1  1.000000   0.000000   0.297872    0.702128   0.924202  0.000000   0.066489
-3  0.000000   1.000000   0.010638    1.989362   0.378989  0.000000  -0.332447
 2  0.000000   0.000000   0.000000    0.000000  -0.656250  1.000000   0.312500
```

Matrix A is ill-conditioned! And the ***unreached space*** is the row and column of the **1.000000**; **simply *zero out* that row and column**, yielding:

```
        1           2          -3        Answer#1:
 1  1.000000   0.000000   0.297872    0.702128   0.924202   0.000000   0.066489
 3  0.000000   1.000000   0.010638    1.989362   0.378989   0.000000  -0.332447
-2  0.000000   0.000000   0.000000    0.000000   0.000000   0.000000   0.000000
```

I suggest the notation $[A]^{-D} = [Ad]$ for the chosen linearly-independent inverse subset of **[A]**.

The – sign on the indices keeps track of what rows and columns aren't used and hence will be zero'd. In this example: **Equation#2** and variable **C** have been bypassed.

[Ad]*[A]=
```
          1.000   0.000   0.298 -> A = 1.0*A + .298*C
          0.000   1.000   0.011 -> B = 1.0*B + .011*C
          0.000   0.000   0.000 -> C =         .000*C
```
…showing how to combine the unknowns.

[A]*[Ad]=
```
          1.000   0.000   0.000 -> Eqn#1= 1.000*Eqn#1
          0.656   0.000  -0.313 -> Eqn#2=  .656*Eqn#1-  .313*Eqn#3
          0.000   0.000   1.000 -> Eqn#3=             1.000*Eqn#3
```
… showing how to combine the equations.
```
Eqn#2= .656*Eqn#1- .313*Eqn#3
                        Eqn#3= 2.095*Eqn#1 -3.194*Eqn#2
```
Actually, at full precision: **Eqn#3= 2.1 *Eqn#1 -3.2 *Eqn#2** as intended.

HAT's overwriter performs the same computation, but in condensed notation:
```
        1       -3       2        Answer#1:
 1   0.924    0.000   0.066       0.702128   = A
 3   0.379    0.000  -0.332       1.989362   = B
-2   0.000    0.000   0.000       0.000000   = C
```

Again: **Equation#2** and variable **C** have been eliminated in **[Ad]** – because their forward partial derivative reduced to zero during the inversion process. The values for **A**, **B**, & **C** exactly satisfy Eqn#1, Eqn#2, and revised Eqn#3 simultaneously – but only because Equation #3 was already in the 2-D inverted subspace.

When the OverWriter returns zeroed rows and columns inside the inverse matrix – HAT has chosen a linearly dependent subset of the solution space to eliminate, **from among several/many (at least two) possible choices**. While it may seem more like *a bother* than *a boon* to return these zero'd values, the fact is that traditional matrix inverters *stop*, providing none of the (hyper)spatial insight that **[A]*[Ai]** does (e.g.: above). **Being able to *revive collapsed solutions* has analytical benefits that shine when solving non-linear problems,** which will be briefly discussed at the end of this paper. There's one benefit that is easy to explain, applies to HAT, and is *amazing* (at least to me):

In solving real-world engineering problems – **vital information** often exists within *what appears to be* (in "casual" observation) *worthless noise*. At the same time, **real-world problems often have *close*, but not *exact*, *partial derivatives*.** Unlike the linearly dependent example above, where the third-pass partial derivative was **0.000000** , commonly the remaining derivatives get smaller and smaller without actually going to zero. Every vector that's inverted is implicitly "a signal", and every vector that isn't inverted is "part of the noise". So, in inversion, a "noise floor" is established that's greater than zero, below which the *fractional contributions to the determinants* will be ignored. *What amazes me* is that, as the determinant of the incoming matrix gets closer to zero (drilling down into the *noise*), the determinant of the inverse grows by a reciprocal amount (becomes an increasingly important *signal*)… which would go to infinity in the limit. So, in the inverse matrix, **the most dominant signals are right next to the noise that was excluded!** If the *noise* is allowed to invert, your answers are likely to be swamped by *nonsense*! For arbitrary problems, at least part of the information supporting the accurate answers resides close to the source of wrong answers; the essence of accurate problem solving is *that harsh*!

…and Mother Nature isn't even trying to deceive you, because she's impartial… which is as close to "fair" as you can reasonably hope to get… mistakes that result will be yours alone - after you've outgrown HAT.

Appendix A has *introductory matrix solver* details.
Appendix B has *a matrix PseudoInverter, OverWriter, & Linear Dependence Eliminator..*
Appendix C: References Hat.exe version 0.50 – presently a matrix-based Polynomial Solver.

For those with keen interest in what lies beyond HAT.exe, consider:

Pseudoinverse System Analysis

Eventually a significant fraction of the world's sensor calibrations will be done using physics models of the sensors – characterizing sensors by adjusting the coefficients of their physics models to fit the data. Part of the elegance of this approach is that, when a sensor fails calibration, there's a direct connection to what went wrong inside the sensor; another part of the elegance is that the understanding of the physics of the device is confirmed to be sufficiently accurate for the present purposes; another part of the elegance is that the knowledge of how the device works is not lost as experts drift away from the project.

Here are the additional concepts:

1. Most physics models are non-linear. Imagine that "solving problems" is about finding your way to the bottom of an "error valley". Linear systems resemble "one big valley" – such that, no matter where you start, in one step you go the very bottom of the only valley… in a "least-squares sense". Non-linear models aren't "one big valley", instead, they're like range of mountains, and *if you plunk yourself down anywhere* & head downhill, you may arrive at the bottom of the wrong valley. An initial guess about the coefficients of your model that puts the system analyzer "in the right valley" avoids a lot of iterating.

2. Given a physics model, numerical partial derivatives are easy to compute. Tweak the coefficients a very small amount, note the resulting changes in all the outputs, divide the output changes by the coefficient changes, and presto: *you have the local numerical partial derivatives*.
The partial derivatives form the **[B]** matrix, and **(the present model Z- the measured Z)** form the **Zerror** vector or **[Zerror]** matrix. When you solve the **[B]:[Zerror]** system for **deltaX**, the **deltaX** vector (which is a linear answer) will probably take you too far… to a place where **X** produces a larger magnitude (length) of **Zerror** than where you started; but keep multiplying **deltaX** by smaller and smaller step sizes, and at some closer range you'll find lower error. Go there and repeat the process of generating the partials.

3. Many physics models are locally linear around their correct solution. Hence, as your **deltaX**'s move the solution farther downhill, the rate of convergence usually accelerates.

In developing your algorithms for Pseudoinverse System Analysis, start with a seemingly simple example to which you already know the answer, e.g.:

$Z = \mathbf{A} * W^{\mathbf{B}}$ **X(1)=A, X(2)=B**

Where the actual answer is $Z = \mathbf{3.0} * W^{\mathbf{2}}$

Using seven datasets: W=[1.,2.,3.,4.,5.,6.,7.] tweak factor for A & B = .0000001

```
Iteration  Step Size         A           B              Zerror
    0      0.            0.00000000  0.00000000    77.537087899921
    1      1.           60.00000010  0.00000000    49.112116631235
     Note: the inverter didn't "bomb out" with B's partials=0.
    2       .235620     38.56795235  0.29329714    40.530903802222
    3       .247580     24.26810100  0.58499904    34.227718152370
    4       .244914     15.12991934  0.87244842    29.660189115189
    5       .243793      9.77620060  1.13994511    25.936697899004
    6       .244955      6.85297078  1.36831328    22.118812610132
    7       .267468      5.17028361  1.56029434    17.914464550170
    8       .442948      3.56893485  1.81154376    12.152431891637
     Note: the rapid convergence once "close":
    9      1.007952      2.83895959  2.02648293     0.688606551725
   10      1.031367      3.00242358  1.99881980     0.104336532734
   11       .999772      2.99999537  2.00000101     0.000026355628
   12       .999999      3.00000000  2.00000000     0.000000000058
   13      1.000966      3.00000000  2.00000000     0.000000000000
```

With a known answer, it's easy to tell when the bottom of the correct valley has been found.

Appendix P, Entry #2, page 49-50, suggests how *pseudoinverse system analysis* and other *high-dimensional mathematical tools* may aid in achieving ***transparent governance.***

Appendix A: An Introductory Matrix Solver

Pages 25 thru 27 are the output of program : "**M1stUse.exe**"=" **M1UseOut-AppendixA.txt**"
Pages 28 thru 31 are the BASIC source code: "**M1stUse.bas**" (compiled by QuickBASIC 4.5)

The output datafile is: **"M1USEOUT.TXT"** as follows:

```
Output of: 'M1stUse.exe'  version 0.40    2011.09.09 JMS

             The program may have errors.
      Input data may have been mis-interpreted.
   USE THIS PROGRAM'S RESULTS ONLY AT YOUR OWN RISK.

Opening file 'MUSEIN.TXT' for input:              Run: 09-09-2011 15:03:54
K-Equations:  3
N-Unknowns :  3
L-Outputs  :  4
kEquations = nUnknowns

[A:Y] will be solved left-to-right.
Input from 'MUseIn.Txt':         (Trailing commas cause read errors.)
        1          2          3
 1    0.3000     1.0000     0.1000     2.2000     0.0000     1.0000     0.0000
 2    1.0000     0.2000     0.3000     1.1000     1.0000     0.0000     0.0000
 3    0.1000     0.2000     1.0000    -0.5000     0.0000     0.0000     1.0000
```

This is like the opening example on page 3,
but rows 1 & 2 have been interchanged (including I) **to exercise row swapping.**

```
Set the noise floor:
ValMin=      0.000000010000000

----- Top of the loop: Reduce Row/column  1: -----
        1          2          3
 1    0.3000     1.0000     0.1000     2.2000     0.0000     1.0000     0.0000
 2    1.0000     0.2000     0.3000     1.1000     1.0000     0.0000     0.0000
 3    0.1000     0.2000     1.0000    -0.5000     0.0000     0.0000     1.0000

Find the largest remaining coefficient in column  1 of [A]:
The abs(max)=           1.0000 at  n= 2
```

No division needed - step skipped.

```
Swapping row  2  with row  1:  [A:Y] becomes:
```

= the example on page 3.

```
        1          2          3
 1    1.0000     0.2000     0.3000     1.1000     1.0000     0.0000     0.0000
 2    0.3000     1.0000     0.1000     2.2000     0.0000     1.0000     0.0000
 3    0.1000     0.2000     1.0000    -0.5000     0.0000     0.0000     1.0000

Subtract row  1  from the other rows using a multiplier:
Reduce   row  2  using multiplier     0.3000 above;   [A:Y] becomes:
        1          2          3
 1    1.0000     0.2000     0.3000     1.1000     1.0000     0.0000     0.0000
 2    0.0000     0.9400     0.0100     1.8700    -0.3000     1.0000     0.0000
 3    0.1000     0.2000     1.0000    -0.5000     0.0000     0.0000     1.0000

Reduce   row  3  using multiplier     0.1000 above;   [A:Y] becomes:
        1          2          3
 1    1.0000     0.2000     0.3000     1.1000     1.0000     0.0000     0.0000
 2    0.0000     0.9400     0.0100     1.8700    -0.3000     1.0000     0.0000
 3    0.0000     0.1800     0.9700    -0.6100    -0.1000     0.0000     1.0000
```

^ At the bottom of the loop: this column has been reduced to the form seen in an identity matrix.

```
----- Top of the loop: Reduce Row/column  2: -----
          1          2          3
 1      1.0000     0.2000     0.3000     1.1000     1.0000     0.0000     0.0000
 2      0.0000     0.9400     0.0100     1.8700    -0.3000     1.0000     0.0000
 3      0.0000     0.1800     0.9700    -0.6100    -0.1000     0.0000     1.0000

Find the largest remaining coefficient in column  2 of [A]:
The abs(max)=           0.9400   at  n= 2
Dividing row  2 by      0.9400,  [A:Y] becomes:
          1          2          3
 1      1.0000     0.2000     0.3000     1.1000     1.0000     0.0000     0.0000
 2      0.0000     1.0000     0.0106     1.9894    -0.3191     1.0638     0.0000
 3      0.0000     0.1800     0.9700    -0.6100    -0.1000     0.0000     1.0000
```

No row swapping needed - step skipped.

```
Subtract row  2  from the other rows using a multiplier:
Reduce   row  1  using multiplier     0.2000 above;   [A:Y] becomes:
          1          2          3
 1      1.0000     0.0000     0.2979     0.7021     1.0638    -0.2128     0.0000
 2      0.0000     1.0000     0.0106     1.9894    -0.3191     1.0638     0.0000
 3      0.0000     0.1800     0.9700    -0.6100    -0.1000     0.0000     1.0000

Reduce   row  3  using multiplier     0.1800 above;   [A:Y] becomes:
          1          2          3
 1      1.0000     0.0000     0.2979     0.7021     1.0638    -0.2128     0.0000
 2      0.0000     1.0000     0.0106     1.9894    -0.3191     1.0638     0.0000
 3      0.0000     0.0000     0.9681    -0.9681    -0.0426    -0.1915     1.0000
```

^ This column has been reduced to the form seen in an identity matrix.

```
----- Top of the loop: Reduce Row/column  3: -----
          1          2          3
 1      1.0000     0.0000     0.2979     0.7021     1.0638    -0.2128     0.0000
 2      0.0000     1.0000     0.0106     1.9894    -0.3191     1.0638     0.0000
 3      0.0000     0.0000     0.9681    -0.9681    -0.0426    -0.1915     1.0000

Find the largest coefficient in column  3 of [A]:
The abs(max)=           0.9681   at  n= 3
Dividing row  3 by      0.9681,  [A:Y] becomes:
          1          2          3
 1      1.0000     0.0000     0.2979     0.7021     1.0638    -0.2128     0.0000
 2      0.0000     1.0000     0.0106     1.9894    -0.3191     1.0638     0.0000
 3      0.0000     0.0000     1.0000    -1.0000    -0.0440    -0.1978     1.0330
```

No row swapping needed - step skipped.

```
Subtract row  3  from the other rows using a multiplier:
Reduce   row  1  using multiplier     0.2979 above;   [A:Y] becomes:
          1          2          3
 1      1.0000     0.0000     0.0000     1.0000     1.0769    -0.1538    -0.3077
 2      0.0000     1.0000     0.0106     1.9894    -0.3191     1.0638     0.0000
 3      0.0000     0.0000     1.0000    -1.0000    -0.0440    -0.1978     1.0330

Reduce   row  2  using multiplier     0.0106 above;   [A:Y] becomes:
          1          2          3
 1      1.0000     0.0000     0.0000     1.0000     1.0769    -0.1538    -0.3077
 2      0.0000     1.0000     0.0000     2.0000    -0.3187     1.0659    -0.0110
 3      0.0000     0.0000     1.0000    -1.0000    -0.0440    -0.1978     1.0330
```

^ This column has been reduced to the form seen in an identity matrix.

```
*** 'm1stUse.exe' - the solution of your input [A:Y] is: ***
[I:X] =
        1          2          3       Answer#1: &  The inverse:
 1    1.0000     0.0000     0.0000     1.0000     1.0769    -0.1538    -0.3077
 2    0.0000     1.0000     0.0000     2.0000    -0.3187     1.0659    -0.0110
 3    0.0000     0.0000     1.0000    -1.0000    -0.0440    -0.1978     1.0330
                                                  Perp#1     Perp#2     Perp#3
```

Note: **The listing order of the equations** doesn't affect Answer#1, but **swaps the columns** of the inverse.
Here the inverse is unchanged because the rows of the appended **I** were re-ordered along with the equations.

```
The Answers for each of your `L-Output` columns:
Answers for column  1: Answer#1
Unknown  1=          1.000000000000
Unknown  2=          2.000000000000
Unknown  3=         -1.000000000000

Answers for column  2:  Perp#1
Unknown  1=          1.076923076923
Unknown  2=         -0.318681318681
Unknown  3=         -0.043956043956

Answers for column  3:  Perp#2
Unknown  1=         -0.153846153846
Unknown  2=          1.065934065934
Unknown  3=         -0.197802197802

Answers for column  4:  Perp#3
Unknown  1=         -0.307692307692
Unknown  2=         -0.010989010989
Unknown  3=          1.032967032967

Done: 09-09-2011 15:03:54.
```

The input datafile: **"M1USEIN.TXT"** first/top dataset used above: (expanded listing: pages 32-33)

```
3,      3,      4
0.30,   1.00,   0.10,   2.20,   0.0,   1.0,   0.0
1.00,   0.20,   0.30,   1.10,   1.0,   0.0,   0.0
0.10,   0.20,   1.00,  -0.50,   0.0,   0.0,   1.0
                                   Unused information follows.
Example vsn 0.50 ~Page  3:  3 equations, 3 unknowns, 4 outputs
```

This is like the opening example on page 3,
but rows 1 & 2 are interchanged (including I) **to demonstrate the row swapping.**

The BASIC source code: **“M1stUse.bas”** follows.
The ~42 lines of code that actually solve [A:Y] are in black bold print.

```
DECLARE SUB PrintAY (nUnk%, MCol%)

REM --------------------------------------------------------------------------
REM Program M1stUse.bas  version 0.50  2011.09.09 Jeff Setterholm

REM Correct numerical examples reduce debug time when writing algorithms.
CLS
CLOSE #14

PRINT "M1stUse.exe    version 0.50          2011.09.09 JMS"
PRINT ""
PRINT "   `Matrix 1st Use` - An Introductory Matrix Solver, "
PRINT "            written in BASIC. Solves [A:Y]. "
PRINT "     The QuickBasic 4.5 source code is provided."
PRINT "M1stUse.exe is limited to:"
PRINT "     1. kEquations=nUnknowns,"
PRINT "     2. Linearly independent equations,  and"
PRINT "     3. Solution left-to-right across the matrix."
PRINT " "
PRINT "M1stUse.exe:"
PRINT "    Reads the first (top) dataset in  'MUSEIN.TXT'"
PRINT "    Writes output/results to:       'M1USEOUT.TXT'"
PRINT ""
PRINT "  ( MUse.exe is a more  powerful   matrix solver,"
PRINT "         but is   more complicated as a result.   )"
PRINT ""
PRINT "         This program may have errors."
PRINT "       Input data may be mis-interpreted."
PRINT "     USE THIS PROGRAM ONLY AT YOUR OWN RISK."
PRINT "  Type 'A' to accept the risks or 'Q' to quit:";
INPUT Accept$
IF Accept$ = "A" GOTO 10
IF Accept$ = "a" GOTO 10
END
10 REM

PRINT "Opening file 'M1USEOUT.TXT' for output:"
OPEN "M1USEOUT.TXT" FOR OUTPUT AS #14

PRINT #14, "Output of: 'M1stUse.exe'  version 0.50    2011.09.09 JMS"
PRINT #14, ""
PRINT #14, "              The program may have errors."
PRINT #14, "      Input data may have been mis-interpreted."
PRINT #14, "   USE THIS PROGRAM'S RESULTS ONLY AT YOUR OWN RISK."
PRINT #14, ""

PRINT "Opening file 'MUSEIN.TXT'   for input:"
PRINT #14, "Opening file 'MUSEIN.TXT' for input:                ";
PRINT #14, USING "     Run: & &"; DATE$; TIME$
OPEN "MUSEIN.TXT" FOR INPUT AS #12

REM ---
REM QuickBASIC 4.5 syntax:
REM   ' :text following an apostrophe is a "Remark" (not compiled);
'     variables ending in % are 16-bit integers;
'     variables ending in # are 64-bit`double precision`floating point numbers;
'     QB4.5 is ~ not case sensitive.
'I use variable names starting with i,j,k,l,m,& n for integers.

INPUT #12, kEqu%, nUnk%, LOut%
PRINT #14, USING "K-Equations: ##"; kEqu%
PRINT #14, USING "N-Unknowns : ##"; nUnk%
PRINT #14, USING "L-Outputs  : ##"; LOut%
IF (kEqu% <> nUnk%) THEN
  PRINT "The number of Equations must equal the number of Unknowns. Halt."
  PRINT #14, "The number of Equations must equal the number of Unknowns. Halt."
  REM STOP
  END
END IF '(kEqu%<>nUnk%)
PRINT #14, "kEquations = nUnknowns"
PRINT #14, ""
PRINT #14, "[A:Y] will be solved left-to-right."

MCol% = nUnk% + LOut%           'MCol%= total number of columns of matrix [A:Y]
DIM AY#(nUnk%, MCol%)

FOR k% = 1 TO kEqu%
  FOR m% = 1 TO MCol%
    INPUT #12, AY#(k%, m%)
  NEXT m%
NEXT k%
PRINT "Closing file 'MUSEIN.TXT'."
CLOSE #12
PRINT #14, "Input from 'MUseIn.Txt':";
PRINT #14, "          (Trailing commas cause read errors.)"
CALL PrintAY(nUnk%, MCol%)

REM Solve AY#[]=[A:Y] left-to-right:

PRINT #14, "Set the noise floor:"
ValMin# = ABS(AY#(1, 1))
FOR n% = 1 TO nUnk%
  FOR m% = 1 TO nUnk%
    IF (ValMin# < ABS(AY#(n%, m%))) THEN
        ValMin# = ABS(AY#(n%, m%))
    END IF
  NEXT m%
NEXT n%
ValMin# = ValMin# / 100000000#
PRINT #14, USING "ValMin=######.###############"; ValMin#
PRINT #14, “”

FOR NextCol% = 1 TO nUnk%
  PRINT #14, USING "----- Top of the Loop: Reduce Row/column###: -----"; NextCol%
  CALL PrintAY(nUnk%, MCol%)

  PRINT #14, USING "Find the largest coeff. in column ## of [A]:"; NextCol%
  ValMax# = ValMin#
  nRowMax% = 0
  FOR nRowTest% = NextCol% TO nUnk%
    IF (ABS(ValMax#) < ABS(AY#(nRowTest%, NextCol%))) THEN
          ValMax#  =    AY#(nRowTest%, NextCol%)
          nRowMax% =         nRowTest%
    END IF
  NEXT nRowTest%
  IF (nRowMax% = 0) THEN
    PRINT "The input equations are linearly dependent."
    PRINT #14, "The input equations are linearly dependent. Halt."
    PRINT "Closing file 'M1USEOUT.TXT'."
    CLOSE #14
    PRINT "Halt."
    END
  END IF '(nRowMax%=0)
  PRINT #14, USING "The abs(max)=     #######.####"; ValMax#;
  PRINT #14, USING "   at  n=##"; nRowMax%

  IF (ValMax# <> 1#) THEN
    PRINT #14, USING "Dividing row ## by ######.####,"; nRowMax%; ValMax#;
    PRINT #14, "  [A:Y] becomes:"
    FOR m% = 1 TO MCol%
       AY#(nRowMax%, m%) = AY#(nRowMax%, m%) / ValMax#
    NEXT m%
    CALL PrintAY(nUnk%, MCol%)
   ELSE
    PRINT #14, "No division needed - step skipped."
    PRINT #14, ""
  END IF '(ValMax#<>1#)

  IF (nRowMax% <> NextCol%) THEN
    PRINT #14, USING "Swapping row ##  with row ##:"; nRowMax%; NextCol%;
    PRINT #14, "   [A:Y] becomes:"
    FOR m% = 1 TO MCol%
      A1# = AY#(nRowMax%, m%)
            AY#(nRowMax%, m%) = AY#(NextCol%, m%)
                                AY#(NextCol%, m%) = A1#
    NEXT m%
    CALL PrintAY(nUnk%, MCol%)
   ELSE
    PRINT #14, "No row swapping needed - step skipped."
    PRINT #14, ""
  END IF '(nRowMax%<>NextCol%)

  PRINT #14, USING "Subtract row ##  from the other rows"; NextCol%;
  PRINT #14, " using a multiplier:"
  FOR n% = 1 TO nUnk%
    IF (n% <> NextCol%) THEN
                                 ValNext# = AY#(n%, NextCol%)
      FOR m% = 1 TO MCol%
         AY#(n%, m%) = AY#(n%, m%) - ValNext# * AY#(NextCol%, m%)
      NEXT m%
      PRINT #14, USING "Reduce   row ## "; n%;
      PRINT #14, USING " using multiplier #####.#### above; "; ValNext#;
      PRINT #14, "  [A:Y] becomes:"
      CALL PrintAY(nUnk%, MCol%)
    END IF '(n%<>NextCol%)
  NEXT n%
NEXT NextCol%
```

```
PRINT #14, "*** 'm1stUse.exe' - the solution of your input [A:Y] is: ***"
PRINT #14, "[I:X] ="
CALL PrintAY(nUnk%, MCol%)
IF (LOut% > 0) THEN
  PRINT #14, "The Answers for each of your `L-Output` columns:"
  FOR L% = 1 TO LOut%
    PRINT #14, USING "Answers for column ##:"; L%
    FOR n% = 1 TO nUnk%
      PRINT #14, USING "Unknown###="; n%;
      PRINT #14, USING " #########.############"; AY#(n%, nUnk% + L%)
    NEXT n%
    PRINT #14, ""
  NEXT L%
END IF '(LOut% > 0)

PRINT #14, USING "Done: & &."; DATE$; TIME$
PRINT "Closing file 'M1USEOUT.TXT'."
PRINT USING "Done: & &            Press escape."; DATE$; TIME$
CLOSE #14
END
REM ---------------------------------------------------------------------------
SUB PrintAY (nUnk%, MCol%)
  SHARED AY#()
  PRINT #14, "   ";
  FOR m% = 1 TO nUnk%
    PRINT #14, USING "######     "; m%;
  NEXT m%
  PRINT #14, " "
  FOR n% = 1 TO nUnk%
    PRINT #14, USING "##"; n%;
    FOR m% = 1 TO MCol%
      PRINT #14, USING "######.####"; AY#(n%, m%);
    NEXT m%
    PRINT #14, ""
  NEXT n%
  PRINT #14, ""
END SUB
---------------------------------------------------------------------------------
```

The input datafile: **“MUSEIN.TXT”**: Used by both “M1stUse.exe” & “MUse.exe”

```
----------------------------------------------------------------------------------------------------------------------------------------------------------------------
3,      3,      4
0.30,   1.00,   0.10,   2.20,   0.0,   1.0,   0.0
1.00,   0.20,   0.30,   1.10,   1.0,   0.0,   0.0
0.10,   0.20,   1.00,  -0.50,   0.0,   0.0,   1.0
                                   Unused information follows.
Example vsn. 0.50 ~Page  3:  3 equations, 3 unknowns, 4 outputs...  [A:Y]
Opening example - but rows 1 & 2 are interchanged (including I)
                          to show row swapping.
                - Row swapping restores the original example
                          and the solution proceeds.
The test case polynomial is: 1.0*X(1) + 2.0*X(2) - 1.0*X(3) = Y
--------------------------------------------------------------
Test datasets for: M1stUse.bas/.exe  version 0.50    2011.09.09 Jeff Setterholm
                          (A simple matrix solver for kEquations=nUnknowns.)
             and: MUse.bas   /.exe  version 0.50
                          (An OverWriting matrix solver.)
                  ^Both these programs relate to Hat.pdf version 0.50

Only the top dataset is read and used.

Note: Trailing commas will cause data misreads.
The testcase polynomial is: 1.0*X(1) + 2.0*X(2) - 1.0*X(3) = Y
--------------------------------------------------------------
3,      3,      4
1.00,   0.20,   0.30,   1.10,   1.0,   0.0,   0.0
0.30,   1.00,   0.10,   2.20,   0.0,   1.0,   0.0
0.10,   0.20,   1.00,  -0.50,   0.0,   0.0,   1.0
                                   Unused information follows.
Example vsn. 0.50 ~Page  3:  3 equations, 3 unknowns, 4 outputs...  [A:Y]
Opening example
The test case polynomial is: 1.0*X(1) + 2.0*X(2) - 1.0*X(3) = Y
--------------------------------------------------------------
3,      3,      4
1.00,   0.20,   0.30,   1.10,   1.0,   0.0,   0.0
0.30,   1.00,   0.10,   2.20,   0.0,   1.0,   0.0
1.14,  -2.78,   0.31,  -4.73,   0.0,   0.0,   1.0
                                   Unused information follows.
Appendix B's TestCase for MUse.exe: Linear Depencence
Example vsn 0.50 ~Page 21:   4 equations, 3 unknowns, 1 output...  [A:Y]
The testcase polynomial is: 1.0*X(1) + 2.0*X(2) - 1.0*X(3) = Y
--------------------------------------------------------------
12, 12, 1
  1.,    1. ,   1.  ,  0.20,  0.20,  0.20,  0.30,  0.30,  0.30 , 0.06,  0.06,  0.06 , 1.1
  1.,    0.3,   0.09,  1.  ,  0.3 ,  0.09,  0.1 ,  0.03,  0.009, 0.1 ,  0.03,  0.009, 2.2
  1.,    0.1,   0.01,  0.2 ,  0.02,  0.002, 1.  ,  0.1 ,  0.01 , 0.2 ,  0.02,  0.002,-0.5
  1.,   -1. ,   1.  ,  0.3 , -0.3 ,  0.3 ,  0.2 , -0.2 ,  0.2  , 0.06, -0.06,  0.06 ,-0.6
  1.,    0.5,   0.25, -1.  , -0.5 , -0.25, -0.3 , -0.15, -0.075, 0.3 ,  0.15,  0.075,-1.2
  1.,   -1. ,   1.  ,  0.  ,  0.  ,  0.  ,  2.  , -2.  ,  2.   , 0.  ,  0.  ,  0.   ,-3.0
  1.,   -1. ,   1.  ,  0.5 , -0.5 ,  0.5 , -2.  ,  2.  , -2.   ,-1.  ,  1.  , -1.   , 2.0
  1.,   -1. ,   1.  ,  0.5 , -0.5 ,  0.5 ,  2.  , -2.  ,  2.   , 1.  , -1.  ,  1.   ,-2.0
  1.,    1. ,   1.  , -0.5 , -0.5 , -0.5 , -2.  , -2.  , -2.   , 1.  ,  1.  ,  1.   , 2.0
  1.,    0. ,   0.  ,  0.5 ,  0.  ,  0.  , -2.  ,  0.  ,  0.   ,-1.  ,  0.  ,  0.   , 3.0
  1.,    1. ,   1.  , -0.5 , -0.5 , -0.5 ,  2.  ,  2.  ,  2.   ,-1.  , -1.  , -1.   ,-2.0
  1.,    1. ,   1.  ,  0.5 ,  0.5 ,  0.5 , -2.  , -2.  , -2.   ,-1.  , -1.  , -1.   , 4.0
                                                    Unused information follows.
       X(1)^1        X(2)^1                  X(3)^1                               Y
 X1X2X3 X1X2X3 X1X2X3 X1X2X3 X1X2X3 X1X2X3 X1X2X3 X1X2X3 X1X2X3 X1X2X3 X1X2X3 X1X2X3:  Y
 ^0^0^0 ^1^0^0 ^2^0^0 ^0^1^0 ^1^1^0 ^2^1^0 ^0^0^1 ^1^0^1 ^2^0^1 ^0^1^1 ^1^1^1 ^2^1^1
```

```
Example vsn 0.50 ~Page 12: 12 of the 27 equations, 12 unknowns, 1 outputs
These are some of the polynomial partial derivatives and outputs of a testcase polynomial.
The test case polynomial is: 1.0*X(1) + 2.0*X(2) - 1.0*X(3) = Y
-------------------------------------------------------------------------------------------
5     3     1
 1.0,  0.2,  0.3,  1.1
 0.3,  1.0,  0.1,  2.2
 0.1,  0.2,  1.0, -0.5
-1.0,  0.3,  0.2, -0.6
 0.5, -1.0, -0.3, -1.2
                                    Unused information follows.
Example vsn 0.50 ~Page  6:  5 equations, 3 unknowns, 1 outputs ...  [B:Z]
More equations than unknowns
The testcase polynomial is: 1.0*X(1) + 2.0*X(2) - 1.0*X(3) = Z
----------------------------------------------------------------
27,    3,    1
    1.0 ,  0.2 ,  0.3 ,  1.1
    0.3 ,  1.0 ,  0.1 ,  2.2
    0.1 ,  0.2 ,  1.0 , -0.5
   -1.0 ,  0.3 ,  0.2 , -0.6
    0.5 , -1.0 , -0.3 , -1.19
   -1.00,  0.00,  2.00, -3.00
   -1.00,  0.50, -2.00,  2.00
   -1.00,  0.50,  0.00,  0.00
   -1.00,  0.50,  2.00, -2.00
    0.00, -0.50, -2.00,  1.00
    0.00, -0.50,  0.00, -1.00
    0.00, -0.50,  2.00, -3.00
    0.00,  0.00, -2.00,  2.00
    0.00,  0.00,  0.00,  0.00
    0.00,  0.00,  2.00, -2.00
    0.00,  0.50, -2.00,  3.00
    0.00,  0.50,  0.00,  1.00
    0.00,  0.50,  2.00, -1.00
    1.00, -0.50, -2.00,  2.00
    1.00, -0.50,  0.00,  0.00
    1.00, -0.50,  2.00, -2.00
    1.00,  0.00, -2.00,  3.00
    1.00,  0.00,  0.00,  1.00
    1.00,  0.00,  2.00, -1.00
    1.00,  0.50, -2.00,  4.00
    1.00,  0.50,  0.00,  2.00
    1.00,  0.50,  2.00,  0.00
                                    Unused information follows.
    X(1)   X(2)   X(3)    Y

Example vsn 0.50 ~Page 11: 27 equations, 3 unknowns, 1 outputs ...  [B:Z]
equations>unknowns; dataset with Y(5) noise.
The testcase polynomial is: 1.0*X(1) + 2.0*X(2) - 1.0*X(3) = Z
-------------------------------------------------------------------------------------------
```

^ This is a partial listing. For the full listing, download:
http://ftp.setterholm.com/PseudoInverse/AppendixA/MUseIn.txt
as well as: /M1stUse.bas,
/M1stUse.txt ,
& /m1stUse.exe

End of Appendix A: An Introductory Matrix Solver

Appendix B: Matrix Solver Details

The BASIC source code of "MUse.bas/.exe:

A Matrix PseudoInverter, OverWriter, & Linear Dependence Eliminator.

http://ftp.setterholm.com/PseudoInverse/AppendixB includes:

```
09/14/2011  09:38 AM            21,297 MUse.bas               … listed here.
09/14/2011  09:39 AM            53,448 MUSE.EXE
09/09/2011  03:08 PM             9,711 MUSEIN.TXT
09/14/2011  09:39 AM             7,785 MUSEOUT-AppendixB.TXT  … listed here.
09/14/2011  07:54 AM             9,119 MUSEOUT-Page6Example.TXT
08/09/2011  08:36 AM             1,205 _StDos.bat
```

Pages 34 thru 37 are **"MUseOut.txt";**
pages 38 thru 47 are : "**MUse.bas**"

"MUseOut.txt":

```
Output of: 'MUse.exe'     version 0.40    Run: 09-14-2011 09:39:38

A Matrix OverWriter & Linear Dependence Eliminator in action...

             The program may have errors.
      Input data may have been mis-interpreted.
   USE THIS PROGRAM'S RESULTS ONLY AT YOUR OWN RISK!

   This output is intended to be useful as 'TestCase Data'
  in writing and debugging your own Matrix OverWriter code
           in your computer language of choice.

Opening file 'MUSEIN.TXT'  for input:                  Run: 09-14-2011 09:00:24
K-Equations:  3
N-Unknowns :  3
L-Outputs  :  1
Your input matrix:          (Trailing commas cause read errors.)
         1            2            3
  1     1.000000     0.200000     0.300000     1.100000
  2     0.300000     1.000000     0.100000     2.200000
  3     1.140000    -2.780000     0.310000    -4.730000
                                                … this is the problem on page 21.
--- Entering: kEquations = nUnknowns; solve directly: ---
             Coefficients: The Outputs
Your input: [    [A]    :   [Y]     ]
   solving: [    [A]    :   [Y]     ]

  yielding: [    [Ai]   :   [X]     ]
      i.e.:  The inverse:The Answers
     ...an '~exact fit' if [A] is linearly independent.

Matrix to be solved:
         1            2            3
  1     1.000000     0.200000     0.300000     1.100000
  2     0.300000     1.000000     0.100000     2.200000
  3     1.140000    -2.780000     0.310000    -4.730000

--- Entering Subroutine OverWriter(): ---

Set the noise floor:
ValMin=     0.000000027800000
```

```
    *** Top of the loop: Iteration  1: ***
          -1           -2           -3
 -1      1.000000     0.200000     0.300000     1.100000
 -2      0.300000     1.000000     0.100000     2.200000
 -3      1.140000    -2.780000     0.310000    -4.730000

The abs(max)=          -2.7800 at  n=  3  m= 2
Det.Product =          -2.780000
Divide row  3 by       -2.7800:
          -1            2           -3
 -1      1.000000     0.200000     0.300000     1.100000
 -2      0.300000     1.000000     0.100000     2.200000
  3     -0.410072     1.000000    -0.111511     1.701439

Swap row  3  with row  2:
          -1            2           -3
 -1      1.000000     0.200000     0.300000     1.100000
  3     -0.410072     1.000000    -0.111511     1.701439
 -2      0.300000     1.000000     0.100000     2.200000

Swap column  2  with column  3:
          -1           -3            2
 -1      1.000000     0.300000     0.200000     1.100000
  3     -0.410072    -0.111511     1.000000     1.701439
 -2      0.300000     0.100000     1.000000     2.200000

Subtract iPivot row  2 from the other rows using a multiplier:
Reduce   row  1  using multiplier     0.2000 above:
          -1           -3            2
 -1      1.082014     0.322302     0.000000     0.759712
  3     -0.410072    -0.111511     1.000000     1.701439
 -2      0.300000     0.100000     1.000000     2.200000

Reduce   row  3  using multiplier     1.0000 above:
          -1           -3            2
 -1      1.082014     0.322302     0.000000     0.759712
  3     -0.410072    -0.111511     1.000000     1.701439
 -2      0.710072     0.211511     0.000000     0.498561

and OverWrite the inverse in column  3   [A:Y] becomes:
          -1           -3            2
 -1      1.082014     0.322302     0.071942     0.759712
  3     -0.410072    -0.111511    -0.359712     1.701439
 -2      0.710072     0.211511     0.359712     0.498561
```

```
   *** Top of the loop: Iteration  2: ***
         -1            -3            2
 -1      1.082014      0.322302      0.071942      0.759712
  3     -0.410072     -0.111511     -0.359712      1.701439
 -2      0.710072      0.211511      0.359712      0.498561

The abs(max)=           1.0820 at  n=  1  m= 1
Det.Product =          -3.008000
Divide row  1 by        1.0820:
          1            -3            2
  1      1.000000      0.297872      0.066489      0.702128
  3     -0.410072     -0.111511     -0.359712      1.701439
 -2      0.710072      0.211511      0.359712      0.498561
No row swapping needed - step skipped.
No column swapping needed - step skipped.

Subtract iPivot row  1 from the other rows using a multiplier:
Reduce   row  2  using multiplier    -0.4101 above:
          1            -3            2
  1      1.000000      0.297872      0.066489      0.702128
  3      0.000000      0.010638     -0.332447      1.989362
 -2      0.710072      0.211511      0.359712      0.498561

Reduce   row  3  using multiplier     0.7101 above:
          1            -3            2
  1      1.000000      0.297872      0.066489      0.702128
  3      0.000000      0.010638     -0.332447      1.989362
 -2      0.000000      0.000000      0.312500     -0.000000

and OverWrite the inverse in column  1   [A:Y] becomes:
          1            -3            2
  1      0.924202      0.297872      0.066489      0.702128
  3      0.378989      0.010638     -0.332447      1.989362
 -2     -0.656250      0.000000      0.312500     -0.000000

   *** Top of the loop: Iteration  3: ***
          1            -3            2
  1      0.924202      0.297872      0.066489      0.702128
  3      0.378989      0.010638     -0.332447      1.989362
 -2     -0.656250      0.000000      0.312500     -0.000000

The input equations are linearly dependent.
  Negative indices indicate dependent rows & columns.
Overwriter inverse zero-ing uses the negative indices.
Salvaging a linearly-independent subset of [Ai] as [Ad]:
          1            -3            2
  1      0.924202      0.000000      0.066489      0.702128
  3      0.378989      0.000000     -0.332447      1.989362
 -2      0.000000      0.000000      0.000000      0.000000
```

```
*** Solver's results: ***
Determinant =          -3.008000
       Rank =           2
         1           -3            2
  1     0.924202     0.000000     0.066489     0.702128  = A
  3     0.378989     0.000000    -0.332447     1.989362  = B
 -2     0.000000     0.000000     0.000000     0.000000  = C
```

|------------------- **[Ai] with** ------------------|-- **Answer#1 --|**

linear dependence eliminated.

```
OverWriter Check: [Ap]*[A] = [I] ? No.
         1           -3            2
  1     1.000000     0.000000    -0.000000
  3     0.656250     0.000000    -0.312500
 -2     0.000000     0.000000     1.000000

OverWriter Check: [A]*[Ap] = [I] ? No.
         1           -3            2
  1     1.000000    -0.000000     0.297872
  3    -0.000000     1.000000     0.010638
 -2     0.000000     0.000000     0.000000

--- Exiting  Subroutine OverWriter(): ---

--- Entering Subroutine ErrorEval(): ---

*** Answers & Error evaluation: ***
Answers for column  1:
Unknown  1=       0.702127659574 =  7.021276595745D-001
Unknown  2=       1.989361702128 =  1.989361702128D+000
Unknown  3=       0.000000000000 =  0.000000000000D+000

Error evaluation for column  1:
Equation:    Ycomputed  -       Yin      =       Yerror
   1:       1.100000000       1.100000000       0.000000000 =  0.000000000D+000
   2:       2.200000000       2.200000000       0.000000000 =  0.000000000D+000
   3:      -4.730000000      -4.730000000       0.000000000 =  0.000000000D+000

                              RMS   error=       0.000000000 =  0.000000000D+000

--- Exiting  Subroutine ErrorEval(): ---

 [Ai]*[A] = [I] ?                  No.
         1            2            3
  1     1.000000    -0.000000     0.297872
  2    -0.000000     1.000000     0.010638
  3     0.000000     0.000000     0.000000
```

… the significance is explained on page 22.

```
 [A]*[Ai] = [I] ?                  No.
         1            2            3
  1     1.000000     0.000000    -0.000000
  2     0.656250     0.000000    -0.312500
  3     0.000000     0.000000     1.000000
```

… the significance is explained on page 22.

```
--- Exiting:  kEquations = nUnknowns ---
Done: 09-14-2011 09:39:38 - closing MUSEOUT.TXT -----------------------------------
```

: “**MUse.bas**”:

```
DECLARE SUB PrintAY (nRows%, mCols%, AYsee#())
DECLARE SUB OverWriter (nUnk%, mCol%, AY#())
DECLARE SUB ErrorEval (kEqu%, nUnk%, mCol%, LOut%, iAX1BZ2%)
DECLARE SUB PrintowAY (nRows%, mCols%, AYsee#(), nUsed%(), mUsed%())
REM ---------------------------------------------------------------------------
REM Program MUse.bas     version 0.50  2011.09.14 Jeff Setterholm

REM BASIC compilers are ubiquitous.
REM Correct numerical examples reduce debug time when writing algorithms.
CLS
CLOSE #14

PRINT "MUse.exe       version 0.50          2011.09.14 JMS"
PRINT ""
PRINT "              `Matrix Use` - a matrix solver. "
PRINT "A Matrix OverWriter & Linear Dependence Eliminator... in action."
PRINT "    Written in BASIC.  Solves [A:Y] =   itself    (equations=unknowns)"
PRINT "                                         -or-                         "
PRINT "                                    = [Bt*B:Bt*Z]    (PseudoInverse). "
PRINT "         The QuickBasic 4.5 source code is provided."
PRINT ""
PRINT "     The output is intended to be useful as 'TestCase Data'"
PRINT "    in writing and debugging your own Matrix OverWriter code"
PRINT "             in your computer language of choice."
PRINT ""
PRINT "MUse.exe:"
PRINT "    Reads the first (top) dataset in 'MUSEIN.TXT'"
PRINT "    Writes output/results to:       'MUSEOUT.TXT'"
PRINT ""
REM --- Cautions & Acknowledgement: ---
PRINT "         This program may have errors."
PRINT "       Input data may be mis-interpreted."
PRINT "     USE THIS PROGRAM ONLY AT YOUR OWN RISK."
PRINT "  Type 'A' to accept the risks or 'Q' to quit:";
INPUT Accept$
IF ((Accept$ <> "A") AND (Accept$ <> "a")) THEN END

PRINT ""
PRINT "Opening file 'M1USEOUT.TXT' for output:"
OPEN "MUSEOUT.TXT" FOR OUTPUT AS #14

PRINT #14, "Output of: 'MUse.exe'     version 0.50    2011.09.14 JMS"
PRINT #14, ""
PRINT #14, "A Matrix OverWriter & Linear Dependence Eliminator in action..."
PRINT #14, ""
PRINT #14, "             The program may have errors."
PRINT #14, "      Input data may have been mis-interpreted."
PRINT #14, "   USE THIS PROGRAM'S RESULTS ONLY AT YOUR OWN RISK!"
PRINT #14, ""
PRINT #14, "   This output is intended to be useful as 'TestCase Data'"
PRINT #14, "  in writing and debugging your own Matrix OverWriter code"
PRINT #14, "           in your computer language of choice."
PRINT #14, ""
REM           --- End C&A. ---
```

```
PRINT "Opening file 'MUSEIN.TXT'   for input:"
PRINT #14, "Opening file 'MUSEIN.TXT'  for input:             ";
PRINT #14, USING "     Run: & &"; DATE$; TIME$

REM ---
REM QuickBASIC 4.5 syntax:
REM   ' :text following an apostrophe is a "Remark" (not compiled);
'     variables ending in % are 16-bit integers;
'     variables ending in # are 64-bit`double precision`floating point numbers;
'     QB4.5 is ~ not case sensitive.
'I use variable names starting with i,j,k,l,m,& n for integers.

OPEN "MUSEIN.TXT" FOR INPUT AS #12            '-- Data input:
    INPUT #12, kEqu%, nUnk%, LOut%
    PRINT #14, USING "K-Equations: ##"; kEqu%
    PRINT #14, USING "N-Unknowns : ##"; nUnk%
    PRINT #14, USING "L-Outputs  : ##"; LOut%
                        mCol% = nUnk% + LOut%    'number of Columns.

  kEqu2% = kEqu%                                 'avoids "variable ailiasing"
  nUnk2% = nUnk%                                 '  in calls to subroutines.

IF (kEqu% = nUnk%) THEN '------------------------ kEquations=nUnknowns:
    DIM AY#(nUnk%, mCol%)                        'Continue with data read:
    FOR n% = 1 TO nUnk%
      FOR m% = 1 TO mCol%
        INPUT #12, AY#(n%, m%)
      NEXT m%
    NEXT n%
    PRINT "Closing file 'MUSEIN.TXT'."
  CLOSE #12                                      'Data read completed.

  PRINT #14, "Your input matrix:          (Trailing commas cause read errors.)"
  CALL PrintAY(nUnk%, mCol%, AY#())              'Print the input matrix:
  PRINT #14, "--- Entering: kEquations = nUnknowns; solve directly: ---"
  PRINT #14, "             Coefficients: The Outputs "
  PRINT #14, "Your input: [    [A]    :   [Y]     ]"
  PRINT #14, "   solving: [    [A]    :   [Y]     ]"
  PRINT #14, ""
  PRINT #14, "  yielding: [    [Ai]   :   [X]     ]"
  PRINT #14, "      i.e.:  The inverse:The Answers "
  PRINT #14, "     ...an '~exact fit' if [A] is linearly independent."
  PRINT #14, ""
  PRINT #14, "Matrix to be solved:"
  CALL PrintAY(kEqu%, mCol%, AY#())

  DIM AiX#(nUnk%, mCol%)                      '-- Solve the equations:
  FOR n% = 1 TO nUnk%
    FOR m% = 1 TO mCol%
      AiX#(n%, m%) = AY#(n%, m%)                 'Saves [A:Y] for use below.
    NEXT m%
  NEXT n%
  CALL OverWriter(nUnk%, mCol%, AiX#())          'Solves [Ai:X] (<-[A:Y])
  IF (LOut% > 0) THEN                            'Evaluate the accuracy:
   iAX1BZ2% = 1
   CALL ErrorEval(kEqu%, nUnk%, mCol%, LOut%, iAX1BZ2%)
  END IF '(LOutputs>0)
```

```
 REM ------
 PRINT #14, "[Ai]*[A] = [I] ?"
 DIM AiA#(nUnk%, nUnk2%)
 FOR n% = 1 TO nUnk%                                  '[Ai:A]=[Ai]*[A]
   FOR m% = 1 TO nUnk%
     FOR nm% = 1 TO nUnk%
       AiA#(n%, m%) = AiA#(n%, m%) + AiX#(n%, nm%) * AY#(nm%, m%)
     NEXT nm%
   NEXT m%
 NEXT n%
 CALL PrintAY(nUnk%, nUnk2%, AiA#())
 ERASE AiA#

 PRINT #14, ""
 PRINT #14, "[A]*[Ai] = [I] ?"
 DIM AAi#(nUnk%, nUnk2%)
 FOR n% = 1 TO nUnk%                                  '[A:Ai]=[A]*[Ai]
   FOR m% = 1 TO nUnk%
     FOR nm% = 1 TO nUnk%
       AAi#(n%, m%) = AAi#(n%, m%) + AY#(n%, nm%) * AiX#(nm%, m%)
     NEXT nm%
   NEXT m%
 NEXT n%
 CALL PrintAY(nUnk%, nUnk2%, AAi#())
 ERASE AAi#

 ERASE AY#
 ERASE AiX#

 PRINT #14, ""
 PRINT #14, "--- Exiting:  kEquations = nUnknowns ---"

ELSE '-------------------------------------------- kEquations<>nUnknowns:
   DIM BZ#(kEqu%, mCol%)                               'Continue with data read:
   FOR K% = 1 TO kEqu%
     FOR m% = 1 TO mCol%
       INPUT #12, BZ#(K%, m%)
     NEXT m%
   NEXT K%
   PRINT "Closing file 'MUSEIN.TXT'."
 CLOSE #12                                            'Data read completed.

 PRINT #14, "Your input matrix:          (Trailing commas cause read errors.)"
 CALL PrintAY(kEqu%, mCol%, BZ#())                    'Print the input matrix:
 PRINT #14, "--- Entering: kEquations <> nUnknowns ---"
 PRINT #14, ""
 PRINT #14, "             Coefficients: The Outputs "
 PRINT #14, "Your input: [   [B]   :    [Z]    ]"
 PRINT #14, "Will solve: [ [Bt*B]  :   [Bt*Z]  ]"
 PRINT #14, "        as: [   [A]   :    [Y]    ]"
 PRINT #14, ""
 PRINT #14, "  yeilding: [   [Ai]  :    [X]    ]"
 PRINT #14, "      i.e.:           :The Answers "
 PRINT #14, "        ...a `least-squares best fit` of [Z]."
 PRINT #14, "          : print: [Bp] = [Ai]*[Bt]"
 PRINT #14, "      i.e.:   The pseudoinverse of [B]"
 PRINT #14, "          : print: [Bp]*[B ] =I ?      and "
 PRINT #14, "          : print: [B ]*[Bp]"
```

```
DIM AY#(nUnk%, mCol%)        'Dimension [A:Y] '      PseudoInverse
FOR n% = 1 TO nUnk%                               '   Morph [A:Y] <- [B:Z]
  FOR m% = 1 TO mCol%                             'in eight lines of BASIC code!
    AY#(n%, m%) = 0#
    FOR K% = 1 TO kEqu%                           '[A:Y]=[ [Bt]*[B] : [Bt]*[Z] ]
      AY#(n%, m%) = AY#(n%, m%) + BZ#(K%, n%) * BZ#(K%, m%)
    NEXT K%
  NEXT m%
NEXT n%                                           '   … an elegant summary!

PRINT #14, ""
PRINT #14, "Matrix to be solved:  (note: [A] = [Bt]*[B] is symmetric)"
CALL PrintAY(nUnk%, mCol%, AY#())
REM Call Gain2(nUnk%, mCol%)
DIM AiX#(nUnk%, mCol%)                          '-- Solve the equations:
FOR n% = 1 TO nUnk%
  FOR m% = 1 TO mCol%
    AiX#(n%, m%) = AY#(n%, m%)                     'Saves [A:Y] for use below.
  NEXT m%
NEXT n%
CALL OverWriter(nUnk%, mCol%, AiX#())          'Solves [Ai:X] (<-[A:Y])
REM Call DeGain2(nUnk%, mCol%)
REM PRINT #14, "*** 'MUse.exe' - Solution: ***"
REM CALL PrintAY(nUnk%, mCol%, AY#())

IF (LOut% > 0) THEN                              'Evaluate the accuracy:
  iAX1BZ2% = 2
  CALL ErrorEval(kEqu%, nUnk%, mCol%, LOut%, iAX1BZ2%)
END IF '(LOut% > 0)

REM ------
PRINT #14, "Computing the pseudoinverse: [Bp]="
DIM Bp#(nUnk%, kEqu%)
PRINT #14, "Unknown    ";
FOR K% = 1 TO kEqu%
  PRINT #14, USING "   Eqn:##    "; K%;
NEXT K%
PRINT #14, ""

FOR n% = 1 TO nUnk%                              '[Bp] = ([Bt]*[B])i * [Bt]
  PRINT #14, USING "####   "; n%;
  FOR K% = 1 TO kEqu%
    Bp#(n%, K%) = 0#
    FOR nm% = 1 TO nUnk%
      Bp#(n%, K%) = Bp#(n%, K%) + AiX#(n%, nm%) * BZ#(K%, nm%)
    NEXT nm%
    PRINT #14, USING "######.######"; Bp#(n%, K%);
  NEXT K%
  PRINT #14, ""
NEXT n%
PRINT #14, ""
```

```
  IF (kEqu% < nUnk%) THEN PRINT #14, "[Bp]*[B] = not [I]"
  IF (kEqu% > nUnk%) THEN PRINT #14, "[Bp]*[B] = [I] ?"
  DIM BpB#(nUnk%, nUnk2%)
  FOR n% = 1 TO nUnk%                              '[BpB]=[Bp]*[B]
    FOR m% = 1 TO nUnk%
      FOR K% = 1 TO kEqu%
        BpB#(n%, m%) = BpB#(n%, m%) + Bp#(n%, K%) * BZ#(K%, m%)
      NEXT K%
    NEXT m%
  NEXT n%
  CALL PrintAY(nUnk%, nUnk2%, BpB#())
  ERASE BpB#
  PRINT #14, ""

  IF (kEqu% > nUnk%) THEN PRINT #14, "[B]*[Bp] = not [I]"
  IF (kEqu% < nUnk%) THEN PRINT #14, "[B]*[Bp] = [I] ?"
  DIM BBp#(kEqu%, kEqu2%)
  FOR K% = 1 TO kEqu%                              '[BBp]=[B]*[Bp]
    FOR k2% = 1 TO kEqu%
      FOR nm% = 1 TO nUnk%
        BBp#(K%, k2%) = BBp#(K%, k2%) + BZ#(K%, nm%) * Bp#(nm%, k2%)
      NEXT nm%
    NEXT k2%
  NEXT K%
  CALL PrintAY(kEqu%, kEqu2%, BBp#())
  ERASE BBp#

  ERASE BZ#
  ERASE AY#
  ERASE Bp#

  PRINT #14, "--- Exiting:  kEquations <> nUnknowns ---"
END IF

PRINT #14, ""
PRINT #14, USING "Done: & & - closing MUSEOUT.TXT"; DATE$; TIME$
PRINT "Closing file 'MUSEOUT.TXT'."
PRINT USING "Done: & &            Press escape."; DATE$; TIME$
CLOSE #14
END 'Program MUse.exe - subroutines follow:

REM ------------------------------------------------------------------------
SUB ErrorEval (kEqu%, nUnk%, mCol%, LOut%, iAX1BZ2%)
  SHARED AiX#()
  SHARED AY#()
  SHARED BZ#()

  PRINT #14, "--- Entering Subroutine ErrorEval(): ---"
  PRINT #14, ""
  PRINT #14, "*** Answers & Error evaluation: ***"
  FOR L% = 1 TO LOut%
    PRINT #14, USING "Answers for column ##:"; L%
    FOR n% = 1 TO nUnk%
      PRINT #14, USING "Unknown###="; n%;
      PRINT #14, USING " #########.############"; AiX#(n%, nUnk% + L%);
      PRINT #14, USING " = ##.############^^^^^"; AiX#(n%, nUnk% + L%)
    NEXT n%
    PRINT #14, ""
    PRINT #14, USING "Error evaluation for column ##:"; L%
    PRINT #14, "Equation:    Ycomputed  -       Yin       =       Yerror"
    RMS# = 0#
    AbsMax# = 0#
    nAbsMax% = 0
    FOR K% = 1 TO kEqu%
      PRINT #14, USING "####:"; K%;
      FitValue# = 0#
      FOR n% = 1 TO nUnk%
        SELECT CASE (iAX1BZ2%)
         CASE IS = 1  'kEquations =  nUnknowns
          FitValue# = FitValue# + AY#(K%, n%) * AiX#(n%, nUnk% + L%)
         CASE IS = 2  'kEquations <> nUnknowns
          FitValue# = FitValue# + BZ#(K%, n%) * AiX#(n%, nUnk% + L%)
        END SELECT
      NEXT n%
      PRINT #14, USING "  ######.#########"; FitValue#;
      SELECT CASE (iAX1BZ2%)
       CASE IS = 1    'kEquations =  nUnknowns
        PRINT #14, USING "  ######.#########"; AY#(K%, nUnk% + L%);
        FitValue# = FitValue# - AY#(K%, nUnk% + L%)
       CASE IS = 2    'kEquations <> nUnknowns
        PRINT #14, USING "  ######.#########"; BZ#(K%, nUnk% + L%);
        FitValue# = FitValue# - BZ#(K%, nUnk% + L%)
      END SELECT
      PRINT #14, USING "  ######.#########"; FitValue#;
      PRINT #14, USING " = ##.#########^^^^^"; FitValue#
      IF ABS(AbsMax#) < ABS(FitValue#) THEN
        AbsMax# = FitValue#
        nAbsMax% = K%
      END IF
      RMS# = RMS# + FitValue# * FitValue#
    NEXT K%
    RMS# = SQR(RMS# / kEqu%)
    PRINT #14, ""
    PRINT #14, "                             ";
    PRINT #14, USING "RMS   error=  ######.#########"; RMS#;
    PRINT #14, USING " = ##.#########^^^^^"; RMS#
    IF nAbsMax% > 0 THEN
      PRINT #14, USING "####:                       "; nAbsMax%;
      PRINT #14, USING "AbsMax error=  ######.#########"; AbsMax#;
      PRINT #14, USING " = ##.#########^^^^^"; AbsMax#
    END IF
    PRINT #14, ""
  NEXT L%
  PRINT #14, "--- Exiting  Subroutine ErrorEval(): ---"
  PRINT #14, ""
END SUB 'ErrorEval()
```

```
REM -------------------------------------------------------------------
SUB OverWriter (nUnk%, mCol%, AY#())              '[A:Y]->[Ai:X]
  PRINT #14, "--- Entering Subroutine OverWriter(): ---"
  PRINT #14, ""
  DIM nUsed%(nUnk%)
  DIM mUsed%(nUnk%)
  DIM SwapColumn#(nUnk%)
  DIM SwapRow#(mCol%)
  DIM Asto#(nUnk%, nUnk%) 'Copy of [A] for evaluating [Ai]*[A], etc.

  nUnk2% = nUnk%                                 'avoids "variable ailiasing"
                                                 '  in calls to subroutines.
  PRINT #14, "Set the noise floor:"
  ValMin# = ABS(AY#(1, 1))
  FOR n% = 1 TO nUnk%
    nUsed%(n%) = -n%
    mUsed%(n%) = -n%
    FOR m% = 1 TO nUnk%
      IF ValMin# < ABS(AY#(n%, m%)) THEN
        ValMin# = ABS(AY#(n%, m%))
      END IF
    NEXT m%
    FOR n2% = 1 TO nUnk%                         'Copy [Asto] <- [A]
      Asto#(n%, n2%) = AY#(n%, n2%)
    NEXT n2%
  NEXT n%
  ValMin# = ValMin# / 100000000#
  PRINT #14, USING "ValMin=######.###############"; ValMin#
  PRINT #14, ""

  DetProduct# = 1#

  FOR NextRowNom% = 1 TO nUnk%         'Solving isn`t necessarily sequential.
    PRINT #14, "   *** Top of the Loop: Iteration ";
    PRINT #14, USING "##:"; NextRowNom%;
    PRINT #14, " ***"
    CALL PrintowAY(nUnk%, mCol%, AY#(), nUsed%(), mUsed%())
    REM Find the largest unused coefficient:
    ValMax# = ValMin#
    nRowMax% = 0
    mColMax% = 0
    FOR nRowTest% = 1 TO nUnk%
      IF (nUsed%(nRowTest%) < 0) THEN
        FOR mColTest% = 1 TO nUnk%
          IF (mUsed%(mColTest%) < 0) THEN
            IF ABS(AY#(nRowTest%, mColTest%)) > ABS(ValMax#) THEN
              ValMax# = AY#(nRowTest%, mColTest%)
              nRowMax% = nRowTest%
              mColMax% = mColTest%
            END IF
          END IF '(mUsed%(mColTest%)<0)
        NEXT mColTest%
      END IF '(nUsed%(nRowTest%)<0)
    NEXT nRowTest%
    IF (nRowMax% = 0) THEN
      PRINT "The input equations are linearly dependent."
      PRINT #14, "The input equations are linearly dependent."
      PRINT #14, "  Negative indices indicate dependent rows & columns."
      PRINT #14, "Overwriter inverse zero-ing uses the negative indices."
      PRINT #14, "Salvaging a linearly-independent subset of [Ai] as [Ad]:"
      FOR n% = 1 TO nUnk%
        IF (nUsed%(n%) < 0) THEN
          FOR m% = 1 TO mCol%    '...eliminating linearly dependent rows
            AY#(n%, m%) = 0#
          NEXT m%
        END IF '(nUsed%(n%)<0)
        IF (mUsed%(n%) < 0) THEN
          FOR n2% = 1 TO nUnk%   '...eliminating linearly dependent columns
            AY#(n2%, n%) = 0#
          NEXT n2%
        END IF '(mUsed%(n%)<0)
      NEXT n%
      CALL PrintowAY(nUnk%, mCol%, AY#(), nUsed%(), mUsed%())
      GOTO 90
    END IF '(nRowMax%=0)

    PRINT #14, USING "The abs(max)=     #######.####"; ValMax#;
    PRINT #14, USING " at  n=##,  m=##"; nRowMax%; mColMax%
    DetProduct# = DetProduct# * ValMax#
    iRank% = NextRowNom%
    PRINT #14, USING "Det.Product =###########.######"; DetProduct#

    NextRow% = mColMax%                         'This is the row to be used.
    nUsed%(nRowMax%) = -nUsed%(nRowMax%)
    mUsed%(mColMax%) = -mUsed%(mColMax%)
    nVarsUsed = NextRowNom%
    nPivot% = mUsed%(mColMax%)         '<- Overwritten row.
    mPivot% = nUsed%(nRowMax%)         '<- Overwritten column.
    IF (ValMax# <> 1#) THEN
      PRINT #14, USING "Divide row ## by ########.####:"; nRowMax%; ValMax#
      FOR m% = 1 TO mCol%
         AY#(nRowMax%, m%) = AY#(nRowMax%, m%) / ValMax#
      NEXT m%
      CALL PrintowAY(nUnk%, mCol%, AY#(), nUsed%(), mUsed%())
     ELSE
      PRINT #14, "No division needed - step skipped."
      PRINT #14, ""
    END IF '(ValMax#<>1#)

    IF (nRowMax% <> nPivot%) THEN
      PRINT #14, USING "Swap row ##  with row ##:"; nRowMax%; nPivot%
      FOR m% = 1 TO mCol%
         A1# = AY#(nRowMax%, m%)
               AY#(nRowMax%, m%) = AY#(nPivot%, m%)
                                   AY#(nPivot%, m%) = A1#
      NEXT m%
      n% = nUsed%(nRowMax%)
           nUsed%(nRowMax%) = nUsed%(nPivot%)
                              nUsed%(nPivot%) = n%
      REM PRINT #14, "nUsed%=", nUsed%(1), nUsed%(2), nUsed%(3)
      CALL PrintowAY(nUnk%, mCol%, AY#(), nUsed%(), mUsed%())
     ELSE
      PRINT #14, "No  row   swapping needed - step skipped."
    END IF '(nRowMax%<>nPivot%)
```

```
    IF (mColMax% <> mPivot%) THEN
      PRINT #14, USING "Swap column ##  with column ##:"; mColMax%; mPivot%
      FOR n% = 1 TO nUnk%
        SwapColumn#(n%) = AY#(n%, mColMax%)
                          AY#(n%, mColMax%) = AY#(n%, mPivot%)
                                              AY#(n%, mPivot%) = SwapColumn#(n%)
      NEXT n%
      m% = mUsed%(mColMax%)
           mUsed%(mColMax%) = mUsed%(mPivot%)
                              mUsed%(mPivot%) = m%
      REM PRINT #14, "mUsed%=", mUsed%(1), mUsed%(2), mUsed%(3)
      CALL PrintowAY(nUnk%, mCol%, AY#(), nUsed%(), mUsed%())
    END IF '(mColMax%<>mPivot%)

    REM eliminate the projected components from all the other equations:
    PRINT #14, ""
    PRINT #14, USING "Subtract iPivot row ## from the other rows"; nPivot%;
    PRINT #14, " using a multiplier:"
    FOR m% = 1 TO mCol%
      SwapRow#(m%) = AY#(nPivot%, m%)
    NEXT m%
    FOR n% = 1 TO nUnk%                     'Clear the space for the overwrite:
      SwapColumn#(n%) = AY#(n%, mPivot%)
    NEXT n%
    FOR n% = 1 TO nUnk%
      IF (n% <> nPivot%) THEN
        PRINT #14, USING "Reduce   row ## "; n%;
        PRINT #14, USING " using multiplier #####.####: "; SwapColumn#(n%)
        FOR m% = 1 TO mCol%
           AY#(n%, m%) = AY#(n%, m%) - SwapColumn#(n%) * SwapRow#(m%)
        NEXT m%
        CALL PrintowAY(nUnk%, mCol%, AY#(), nUsed%(), mUsed%())
      END IF '(n%<>nPivot%)
    NEXT n%
    PRINT #14, USING "and OverWrite the inverse in column ## "; mPivot%
    FOR n% = 1 TO nUnk%
      AY#(n%, mPivot%) = AY#(n%, mPivot%) - SwapColumn#(n%) / ValMax#
    NEXT n%
    AY#(nPivot%, mPivot%) = 1# / ValMax#
    PRINT #14, "  [A:Y] becomes:"
    CALL PrintowAY(nUnk%, mCol%, AY#(), nUsed%(), mUsed%())
  NEXT NextRowNom%

90 PRINT #14, "*** Solver's results: ***"
  PRINT #14, USING "Determinant =############.######"; DetProduct#
  PRINT #14, USING "       Rank =############"; iRank%
  CALL PrintowAY(nUnk%, mCol%, AY#(), nUsed%(), mUsed%())

  PRINT #14, "OverWriter Check: [Ai]*[A] = [I] ?"
  DIM AAi#(nUnk%, nUnk2%)
  FOR n% = 1 TO nUnk%                              '[AiA]= [A]*[Asto]
    FOR m% = 1 TO nUnk%
      FOR nm% = 1 TO nUnk%
        AAi#(n%, m%) = AAi#(n%, m%) + AY#(n%, nm%) * Asto#(nm%, m%)
      NEXT nm%
    NEXT m%
  NEXT n%
  CALL PrintowAY(nUnk%, nUnk2%, AAi#(), nUsed%(), mUsed%())
  ERASE AAi#
  PRINT #14, ""
  PRINT #14, "OverWriter Check: [A]*[Ai] = [I] ?"
  DIM AiA#(nUnk%, nUnk2%)
  FOR n% = 1 TO nUnk%                            '[AAi]= [Asto]*[A]
    FOR m% = 1 TO nUnk%
      FOR nm% = 1 TO nUnk%
        AiA#(n%, m%) = AiA#(n%, m%) + Asto#(n%, nm%) * AY#(nm%, m%)
      NEXT nm%
    NEXT m%
  NEXT n%

  CALL PrintowAY(nUnk%, nUnk2%, AiA#(), nUsed%(), mUsed%())
  ERASE AiA#

  ERASE nUsed%
  ERASE mUsed%
  ERASE SwapColumn#
  ERASE SwapRow#
  ERASE Asto#
  PRINT #14, "--- Exiting  Subroutine OverWriter(): ---"
  PRINT #14, ""
END SUB 'OverWriter()

REM ---------------------------------------------------------------------------
SUB PrintAY (nRows%, mCols%, AYsee#())
  PRINT #14, "    ";
  FOR n% = 1 TO nRows%
    IF (n% <= mCols%) THEN PRINT #14, USING "######       "; n%;
  NEXT n%
  PRINT #14, " "
  FOR n% = 1 TO nRows%
    PRINT #14, USING "###"; n%;
    FOR m% = 1 TO mCols%
      PRINT #14, USING "######.######"; AYsee#(n%, m%);
    NEXT m%
    PRINT #14, ""
  NEXT n%
  PRINT #14, ""
END SUB 'PrintAY()

REM ---------------------------------------------------------------------------
SUB PrintowAY (nRows%, mCols%, AYsee#(), nUsed%(), mUsed%())
  PRINT #14, "    ";
  FOR m% = 1 TO nRows%
    IF (m% <= mCols%) THEN PRINT #14, USING "######       "; mUsed%(m%);
  NEXT m%
  PRINT #14, " "
  FOR n% = 1 TO nRows%
    PRINT #14, USING "###"; nUsed%(n%);
    FOR m% = 1 TO mCols%
      PRINT #14, USING "######.######"; AYsee#(n%, m%);
    NEXT m%
    PRINT #14, ""
  NEXT n%
  PRINT #14, ""
END SUB 'PrintowAY()
```

End of Appendix B: Matrix Solver Details

Appendix C: Hat.exe - Use

Limited to **A Matrix-based Polynomial Solver** for now.

http://ftp.setterholm.com/PseudoInverse/AppendixC includes:

```
08/29/2011 11:42 AM  586,240 HAT.exe       - the program.
```

```
08/29/2011 11:03 AM    5,823 HatIn.csv     - the input.
```

Look at 'HatIn.csv' in an ASCII text editor to get a sense of how input datasets are organized.
Hat.exe reads only your first (top) dataset in HatIn.csv.

```
08/29/2011 11:42 AM   30,007 HatReport.txt – the detailed output.
```

'HatReport.txt' provides a good example of what 'Hat.exe' can do ***in the blink of an eye.***

```
08/29/2011 11:42 AM    1,989 HatOut.csv    - reorders input data
```

Look at 'HatOut.csv' in a spreadsheet program.

```
08/29/2011 09:49 AM      225 _Run-Hat.bat
   For use by DOS-literate people. Launches the program
     & follows up by displaying HatReport.txt in the screen window.
```

The opening disclaimers of "HAT.exe" version 0.40 – is an unpleasant read:

```
HAT.exe is an experimental piece of scientific software.
 > A sample`HatIn.csv`file was available with this software.
   with several datasets therein.

 > Presently, ONLY POLYNOMIAL-BASED SOLVING IS ACCESSIBLE.
   `HatReport.txt` has the useful results.
   `HatOut.csv` is for-now only useful for re-ordering data.

    Use `MUse.exe` for non-poly problems (See Appendix B).

   HAT reads only the first (topmost) dataset.
 > The manner in which the software might respond to errors
   in your `HatIn.csv` input file is unknown.
 > The program was created on an AMD Athlon 64 processor in
   a Windows XP environment using Absoft`s ProFortran 9.0.
   Whether or not this program will run properly on your
   particular computer is unknown to me.
 > Although not intentional on my part, there may be errors
   in the computational results.

 >           THIS PROGRAM IS POSTED ON THE WEB
        WITHOUT GUARANTEES OR WARRANTIES OF ANY KIND,
               including, but not limited to,
            fitness for any particular purpose.

 > If YOU ACCEPT ALL THE RISK(S) of running the program:
              type A to accept the risk(s) and continue.
   Otherwise, type Q to  quit. :
```

End of Appendix C: Hat.exe - Use

Appendix P: Philosophy

Entry #1: (Referenced on page 9)

Without a hyper-dimensional way of understanding how "unknowns" relate to "observations", it's easy to be close-to-clueless about how "real world" problems might be solved. Science, Math, and Engineering education & experience have produced people who, by *intense focus* in understanding their disciplines, were/are profoundly capable problem solvers. If the world's social problems are going to be solved peacefully, then **humanity needs people who are *intensely focused problem solvers* within the various social disciplines**..."Your mission – should you choose to accept it – ".

Trusting the ***invisible mental gears*** of "the next politician" will not take the world to social harmony. **We need transparent governance decision models.** Living under a tyrant, 10 years can seem like an eternity – if you're lucky enough to survive. Solve problems now, while you have a chance, or *pay for not solving them* later. Allowing me some poetic license: "**There are only two kinds of people: Engineers & Victims.**" HAT lies *at the beginning* of a path to learning how to create transparent governance decision models which may benefit almost everyone.

Somebody – anybody – anywhere in the world - please go for it!

Entry #2: (Referenced on page 24)

On the social side, I consider it likely that *pseudoinverse system analysis* will be part of the analytical mix that creates *transparent governance models*, helping in innumerable ways, including:

1. By the comparatively simple and robust access that it offers for exploring parameter identifications in high-dimensional non-linear problem spaces.

2. By de-mystifying the very idea of being able to find accurate answers in hyperspaces. Citizens the world over may begin to *expect*, if not *demand*, that presently-funded experts begin to provide stable, long-term solutions to the governance problems *that are theirs to solve*, particularly solutions to the social problems that have plagued humanity for hundreds of years. **Trusting the *invisible mental gears* of "the next politician" has not and never will take the world to enduring social harmony. Let's try to create transparent governance decision models.** Maybe the models won't work either, but an old Army manual characterized plans in a thoughtful way:

"A *bad plan* is better than *no plan*."

3. By recognizing that *every family in America* is a "special interest group" which should have an equal amount of weight in the search for balanced congressional "answers". "The average American family**"** *is falling apart before our very eyes*; do the majority of our mentally-unaided politicians even dare to care? **For the time being: greed rules at the national level, eh?**

The idea of "a transparent ethical compass" that works in hyperspace has allure.

4. *The corruption of human minds by wealth & power* is a commonly recognized pitfall; trusting "invisible mental gears" as "leadership mechanisms" = a bad plan. "Evolving ***transparent*** mitigations of human pitfalls" is a grand vision.

5. An integral part of achieving *transparent governance* involves arriving at a shared comprehension of *the rules that constrain and empower us* – i.e. our Laws. Bright and ambitious young Americans *have been drawn to* ***the rules*** *like a professional magnet for scores of years,* but a significant fraction of lawyers resemble *loose cannons rolling around the deck of a ship,* contributing – in a major way – to financial uncertainties and financial losses for the rest of society. There's no good reason why "a nation's rules" should be the foundation of widespread parasitic professional conduct.

6. More dimensions are involved in the tradeoffs of governance decisions than any one mind can intuitively harmonize. Hyperspace math, *in various forms*, will aid our ***shared discernment***. This document is a piece of the puzzle. As one example of other *various forms* of high-dimensional mathematics: *Linear Programming* is a mathematical tool used to efficiently allocate manufacturing resources within an enterprise.(See Wikipedia.)

7. Both eloquent and "invisible" intentions led to the American Civil War in 1861; the speeches and writings of Thomas Jefferson, America's third President, come to mind. Jefferson was *the philosophical guiding light* of the Confederacy during the war, but, none-the-less, the Jefferson Memorial in Washington D.C. stands as a national monument to his eloquence and influence. Even after giving our third President the benefit of the doubt - that he meant well - ***if there need be proof*** **that "great speeches", and/or "great politicians" are a suspect means of <u>assuring</u> social harmony, Thomas Jefferson's example *provides the proof*.** Adolf Hitler also delivered "great speeches" in his day, but events subsequently revealed Hitler's "invisible" intentions, which harmed/killed millions of people.

8. Years ago someone concluded that: **'The purpose of companies is to utilize people's strengths and make their weaknesses irrelevant.'**

Here's a candidate statement of purpose:
"The purpose of *transparent governance* is to provide a shared & predictable political framework within which individuals and organizations can <u>plan for the future</u>, and to instruct our political leaders in how our society presently functions."

It remains to be seen whether or not *transparent governance* can be achieved.
Deciding ***clearly*****:** "What <u>is</u> us." and "What <u>is</u> <u>not</u> us." will be difficult.
In systems with many dimensions, gems are the neighbors of noise.
It's no wonder that "invisible mental gears" are so challenged by reality.

End of Appendix P: Philosophy

References & Acknowledgements:

"A First Course in Linear Algebra" by Daniel Zelinsky, Academic Press, 1973.
This is an ideal textbook for people who prefer to learn math using intuition and examples.

The Flight Simulation Engineers at McDonnell Douglas, St. Louis (1976-1978).
Within the simulation group, extremely efficient codes for solving problems 20 times a second were the-order-of-the-day; everyone helped everyone else become more skilled at efficient problem solving. Within that talented group of people, there *seemed to be* no lower limit on how compact source codes could become, and there *seemed to be* no lower limit on how quickly a given problem could be solved… when given further thought.

Honeywell's Systems & Research (S&RC), Minneapolis (1978-1984).
(at Ridgway Parkway)
Honeywell had a building full of multi-disciplinary experts who were *as collegial as* the flight simulation engineers at McDonnell Douglas. Within S&RC, **Dr. Gunter Stein** taught me that:

$$[A]^{-P} = ([A]^{T} * [A])^{-1} * [A]^{T}$$

I knew, the instant that Gunter wrote down the equation, that my professional life had just experienced a major empowerment. (I had seen pseudoinverse being used in a very simple control system solution at McDonnell Douglas, but hadn't begun to grasp the scope of the subject.)

Absoft Corporation's ProFortran 9.0 & **William Mitchell's F90GL.**
For the last seven years I've programmed using Absoft's version 9.0 Fortran compiler and the OpenGL (graphics) interface to Fortran provided by Dr.William Mitchell of NIST. The stability of the programming environment and the power of the graphics are a marvel. Bravo.

~Apologies:

1. I haven't been trained as a teacher, so knowing "how to teach" isn't my specialty. I suggest, however, that teaching can be parsed into two subsets: "**How to Teach**" and "**What to Teach**". Consider this paper an exposition on "What to Teach" to empower bright 9th graders to progress into hyperspace analytics. I invite anyone to figure out "how to teach" the material; I would enjoy the opportunity to help with the task. (The source codes and examples in Appendices A and B reveal the mechanics of the computations described on pages one through nine of this document.)

2. Pseudoinverse System Analysis isn't part of HAT because I'm not aware of how to exercise an (your) externally-defined system simulation model – efficiently - from within "HAT.exe"

3. Almost no visualizations are included in this paper, despite having created quite a few (each of which made little intuitive sense to me). In general, many real problems naturally lend themselves to visualizations - demonstration of results in a visual context. Visualizations easily access intuition, whereas numbers alone are, at best, more narrowly intuitive. **Strive to have a personal programming environment that allows you to code your own powerful algorithms and to create your own first-rate 3-D (stereo) dynamic graphic visualizations.** Visually-based analytical exploration is a hoot! "Homogeneous Transforms" (4x4 matrices with special properties) are the key to understanding the math of perspective & 3-D visualization, because you can then efficiently do *projection*; This is yet another example of *brilliant results* produced by scientists whose names may be unfamiliar to you. Expanding homogeneous transforms into hyperspace is likely to be fruitful; e.g.: with some thought, 4-D spaces can probably be *projected* at will onto 3-D subspaces for stereo viewing.

Employment sought:

I'm an unemployed STEM professional **looking for paid work** – ***working for everyone*** in the meantime, at my own expense… simplifying technical understandings to their essentials and providing thoughtful alternatives to "how we do business". Patents - ref: www.uspto.gov - advanced search: IN/Setterholm-Jeffrey-M

My contact information:

Jeffrey M. Setterholm
8095 230th St. E.
Lakeville, Minnesota 55044-8287
USA

This document has a wealth of insights about "what to teach" to *mathematically empower* analytically-inclined young people.

"How to teach" these insights is now the greater challenge.

Review of Contents **Page(s)**

This concludes Hat.pdf, version 0.50