HP Forum Archive 16

How does ill-conditioning happen? (long)

Message #1 Posted by ***Rodger Rosenbaum*** *on 18 May 2006, 3:37 a.m.*

In this post, I will show in detail just how ill-conditioning does its dirty work, and how the SVD can eliminate the problem.

First I'll talk a little about vectors and vector spaces. This may be familiar ground to some, but I'm going to do it in the interest of a coherent presentation.

To keep it simple and fairly easy to visualize, I'm going to talk about vectors in a 3 dimensional space with 3 orthoganal axes (cartesian coordinates). The usual convention is that the components of a 3-element vector will be the x, y, and z components, like this:

[x y z]. We can arrange 3 such vectors as the 3 columns of a 3x3 matrix, like this:

```
[[1 2 3]                                                      [[X1]
 [4 5 6]    Notice that if we postmultiply this matrix by a vector  [X2] , the result is a
 [7 8 9]]                                                      [X3]]
```

linear combination of the 3 column vectors of the matrix. Working out the details, we see that:

```
[[1 2 3]    [[X1]          [[1]        [[2]         [[3]
 [4 5 6]  *  [Y1]   =  X1 *  [4]  + Y1 *  [5]  +  Z1 *  [6]  ; that is, we are computing
 [7 8 9]]    [Z1]]          [7]]        [8]]         [9]]
```

a new vector which is X1 times the first column plus Y1 times the second column plus Z1 times the third column.

If we have 3 vectors [X1 X1 Z1], [X2 Y2 Z2] and [X3 Y3 Z3], such that every vector in 3-space can be generated as a linear combination of them, then the 3 vectors are said to *span* the space, and they form a *basis* for the space. An example of 3 vectors with this property would be [1 0 0], [0 1 0] and [0 0 1]. An example of 3 vectors which don't span 3-space would be [1 0 0], [0 1 0] and [1 1 0]; what is missing here is a component in the Z direction. In this last case the 3 vectors span the XY plane only. The part of N-space not spanned by the columns is called the null space. We can tell when a matrix has a non-empty null space by whether or not any of its singular values are zero. When we're using floating point arithmetic, we can say that a matrix "almost" has a non-empty null space if any of its singular values are very small.

If we have a 3x3 matrix, the space spanned by the columns is called the column space. If and only if the 3 columns are linearly independent (the matrix is not singular) do they span all of 3-space.

Given an n x n matrix A, it is always possible to find an n-element vector which postmultiplies A and gives an all zero result; the vector of all zeroes will always produce this result. But this isn't very interesting and therefore is called the trivial case. If there exists a vector which *isn't* all zeroes, but gives an all zero result when it postmultiplies A, then the 3 column vectors are *not* linearly independent and the matrix is singular.

```
                            [[1 0 1]                                  [[ 1]    [[0]
  For example, the matrix A =  [0 1 1]  is singular because A *  [ 1]  =  [0] , but the
                             [0 0 0]]                                  [-1]]    [0]]
```

vector which postmultiplies A is not all zeroes. This is to be expected because the 3rd column is the sum of the first two columns. But now notice this; if we find a vector which postmultiplies a matrix and gives an all zero result, then *any multiple* of the vector will give the same result. For example:

```
    [[ 1000000]    [[0]
 A *  [ 1000000]  =  [0]
     [-1000000]]    [0]]
```

Now consider what happens if we perturb the A matrix a little and then postmultiply by the two vectors we just used:

```
[[1.001 0 1]     [[ 1]    [[.001]          [[1.001 0 1]     [[ 1000000]    [[1000]
 [  0   1 1]  *   [ 1]  =  [  0 ]   and     [  0   1 1]  *   [ 1000000]  =  [  0 ]
 [  0   0 0]]     [-1]]    [  0 ]]          [  0   0 0]]     [-1000000]]    [  0 ]]
```

The linear combination of the columns in the first two multiplications added up to zero because the postmultiplying vector was created in just such a way as to make that happen. But in the second two multiplications, there was only the one perturbation, so there wasn't anything to cancel it in the final sum. Thus, the perturbation got magnified by the factor of 1E6 which was the multiple of [[1 1 -1]]T used in the last case. We'll see how this figures in later.

Since postmultiplying a matrix A by a vector x is forming a vector which is the linear combination of the columns of A given by the elements of x, we see that when we solve the system:

A * x = B

we are seeking to find a linear combination of the columns of A which will equal B. If a solution x exists, then the elements of x are just the multipliers of the columns of A such that the sum of the multiplied columns adds up to B.

Suppose we try to solve the system A * x = [[0 0 0]]T (this is called the homogeneous case). What we are asking is, "what linear combination of the columns of A equals a vector of all zeroes?". Now the definition of a non-singular matrix is that there is no linear combination of the columns which adds to all zeroes (except the trivial case where the post multiplier is [[0 0 0]]T). Therefore, there is no (non-trivial) solution to the homogeneous case if the matrix is non-singular. If the A matrix is singular, there are many solutions because as we saw above, if we know a (non-trivial) vector which postmultiplies a matrix and gives a result of all zeroes, then any multiple of that vector also gives an all zero result.

Imagine we have a solution x1 such that A * x1 = [[0 0 0]]T, and another solution x2 such that A * x2 = B, B = [[i j k]]T, where not all of i, j and k are zero. Then the solution vector x2 will postmultiply A and give a result B, but so will x2 plus any multiple of x1 since:

A * x2 = B and A * (m * x1 + x2) = (A * m*x1) + (A * x2) = ([[0 0 0]]T) + (A * x2),

because A times any multiple of x1 will give all zeroes, and a vector of all zeroes added to B is still just B.

```
                        [[5 4 5]
  Consider the matrix A =  [4 8 4] .  This matrix is singular because the 1st and 3rd
                         [5 4 5]]
```

columns are identical. But let's postmultiply it by [[1 1 1]]T. The result is [[14 16 14]]T. Suppose we try to solve the system A * x = [[14 16 14]]T. We already know that a solution of this system is [[1 1 1]]T, but the B/A method won't work because A is singular; the INVERSE(A)*B method fails for the same reason. The RREF method gives [[2 1 0]]T; let's test it. Postmultiplying A by [[2 1 0]]T gives [[14 16 14]]T, so it is indeed a solution.

Since A is singular, there should be an infinite number of solutions to the homogeneous case, A * x = [[0 0 0]]T, and they are all multiples of each other. And, in fact, they are all vectors in the null space of A. One such vector is [[1 0 -1]]T. If we multiply out A * [[1 0 -1]]T, we do indeed get [[0 0 0]]T. So, we should be able to find many solutions to A * x = [[14 16 14]]T by adding any multiple of [[1 0 -1]]T to [[1 1 1]]T. If we add [[1 0 -1]]T to [[1 1 1]]T, we get [[2 1 0]]T, the solution found by the RREF method. If we add
1000000 * [[1 0 -1]]T to [[1 1 1]]T, we get [[1000001 1 -999999]]T, and if we postmultiply A by this, sure enough we get [[14 16 14]]T.

For the rest of this post, I'm going to assume the reader has a directory with the variables AA, BB, A, AP, B, U, V, S and the various little programs found in some other posts about the SVD. In the AA variable, store Valentin's original matrix; in BB, store the associated column vector. In variable A, store the 3x3 A matrix I've just been discussing, and in B, store [[14 16 14]]T.

Now, let's perturb the A matrix. Type Pi RDZ A RANM RANM 1E6 / A +. This should give:

```
[[5.000001 3.999991 5.000003]
 [4.000005 8.000005 4.000006]
 [4.999993 3.999994 4.999992]]
```

Store this in AP, and recall AP. Execute ->SV, which should be in your menu; it's one of the little programs I mentioned (or you can type SVD). The stack contains the results of the SVD, namely U V S; store these in the corresponding variables (watch the order).

This matrix, AP, is no longer singular, and therefore theoretically has an empty null space, but because its smallest singular value is so small (about 1.5E-6), it is useful to assume that the last row of V is acually in the (approximate) null space.

Let's try to solve the system AP * x = B; since the perturbed matrix AP is no longer singular, and has a condition number of about 10E6, we should be able to solve it with any method. Executing B AP /, we get a solution vector [[6.33..., .999...,-4.33...]]T, considerably different from the expected [[1 1 1]]T. Let's try the INVERSE(AP)*B method; we get the same result. And the RREF and LSQ methods also give that same result.

To see what's going on, lets use the SVD to find the inverse of AP. On the HP49G+, the S vector from the SVD is a vector of the singular values. Let a lower case plus numeral denote the individual singular values. That is, s1 is the largest, s2 the next smaller, and s3 the smallest. In another post I described how the inverse of a matrix can be derived from the SVD like this:

INVERSE(AP) = TRANSPOSE(V) * DIAG(1/s1, 1/s2, 1/s3) * TRANSPOSE(U), where DIAG(1/s1, 1/s2, 1/s3) is a matrix with the reciprocals of the singular values on the main diagonal, and zeroes elsewhere. Since matrix multiplication is distributive over addition, we can rewrite this as:

```
                [[1/s1  0   0 ]
(TRANSPOSE(V) *  [  0   0   0 ]  * TRANSPOSE(U)
                 [  0   0   0 ]]


                [[  0   0   0 ]
+TRANSPOSE(V) *  [  0  1/s2 0 ]  * TRANSPOSE(U)
                 [  0   0   0 ]]


                [[  0   0   0 ]
+TRANSPOSE(V) *  [  0   0   0 ]  * TRANSPOSE(U) )
                 [  0   0 1/s3]]
```

Now we're getting to the heart of the matter. The result of multiplying INVERSE(AP) * B will be given by premultiplying B by each of the 3 terms just above and adding the products. But look at that last term; The diagonal matrix in the middle of the term contains the reciprocal of the smallest singular value, a very large number if the singular value is very small. If I multiply out:

```
               [[  0   0   0 ]
TRANSPOSE(V) *  [  0   0   0 ]  * TRANSPOSE(U) * B
                [  0   0 1/s3]]
```

I get [[5.333...,6.666E-7...,-5.333...]]T. This looks suspiciously like a multiple of the vector [[1 0 -1]]T which was in the null space of A. This means that if we post multiply AP by this vector, we should get something close to all zeroes. In fact, we get approx. [[-8E-6, -5.8E-11, 8E-6]]T.

Now let's multiply the first two terms above times B:

```
                [[1/s1  0   0 ]
(TRANSPOSE(V) *  [  0   0   0 ]  * TRANSPOSE(U)
                 [  0   0   0 ]]
                [[  0   0   0 ]
+TRANSPOSE(V) *  [  0  1/s2 0 ]  * TRANSPOSE(U) ) * B
                 [  0   0   0 ]]
```

If we do this we get [[1.00000343..., .999994, 1.00000356...]]T; this is very close to the solution we expected to get, [[1 1 1]]T, and if we add the vector that came from the third term above, namely [[5.333...,6.666E-7...,-5.333...]]T, we get [[6.33..., .999..., -4.33...]]T, which is the solution all the traditional methods gave above.

If we multiply A * [[6.33..., .999...,-4.33...]]T, we get [[13.99..., 15.99..., 13.99...]]T, slightly perturbed from the [[14 16 14]]T we expect. It's that last term ([[5.333...,6.666E-7...,-5.333...]]T) in the 3-term inverse expression above that causes the trouble. It's big because of the reciprocal of the smallest singular value in the diagonal matrix, so it perturbs our solution away from the expected [[1 1 1]]T to [[6.33..., .999..., -4.33...]]T. But since it's in the "almost null space" of the matrix AP, it add almost nothing (it would add exactly zero if it were in a "true" null space) to the product of AP * x; its presence hardly changes the product away from the B we expect.

It's because that term associated with the small singular value ends up being big in the expression (it's the 1/s3 value, in this case) for the inverse, that any perturbations in the original A matrix are magnified. And the smaller the singular value (which means that the condition number is higher), the larger 1/s3 becomes, and the more any perturbations in A are magnified.

So what can we do? Since the terms involving very small singular values in the inverse expression are in the "almost null space", they add values near zero to the product A * x and contribute little to the solution. The thing to do is to just don't include those terms in the inverse expression. In our case, this means just use the first two terms and drop the last one. If a particular A matrix has more than one small singular value, then drop the terms associated with all the really small singular values.

Let's go back to our original A matrix:

```
    [[5 4 5]
A =  [4 8 4]
     [5 4 5]]
```

This system, A * x = B can't be solved by any of the traditional methods, but the SVD method can solve it (sometimes LSQ on the HP49G+ can solve these systems, but never if there is more than one small singular value. It does work in this case.). Use the PSU program from another post. Retype the unperturbed A and store it in its variable. Recall A, press ->SV and store U V S in their respective variables. Press 1 PSU B * (the {1 PSU} says to delete 1 singular value, starting from the smallest, in computing the inverse) and get:

[[.999999999996 1.00000000001 .999999999996]]T

You can't do much better than that, especially considering that the condition number of the A matrix is infinite!

For a final two examples, let's play with Valentin's original nearly singular A matrix, which should be stored in the variable AA. Press Pi RDZ AA RANM 1e10 / AA +. Store this perturbed matrix in A; recall A and press ->SV, then store U V S in their respective variables.

Press BB A / to get a traditional solution for a very slightly perturbed A matrix. The solution delivered by the HP49G+ is:

```
[[-.425428235758]
 [ 1.00017970011]
 [-.270939093345]
 [-.278200434138]
 [ .999118037445]
 [ 2.27818077533]
 [ 3.67424980265]]
```

This is a substantial error for a perturbation of the AA matrix on the order of 1E-10. Now let's see what the SVD can do. Press 1 PSU BB * and see:

```
[[.9976...]
 [1.000...]
 [.9979...]
 [.9979...]
 [.9999...]
 [1.002...]
 [1.004...]]
```

Given that the original AA matrix was only accurate to 2 digits, this result rounds to:

[[1 1 1 1 1 1 1]]T

Modify the {1 2} element of the AA matrix by appending ten 1's, so the matrix becomes extremely near singular and store the result in the A variable. Recall A and press ->SV and store U V S in their respective variables. Recall A and postmultiply by [[1 1 1 1 1 1 1]]T; store the result in B. Press 1 PSU B * and see:

```
[[.9976...]
 [1.000...]
 [.9979...]
 [.9979...]
 [.9999...]
 [1.002...]
 [1.004...]]
```

the same result as above. In this case, LSQ doesn't do well at all, nor does RREF. The B/A method does the best, which isn't all that good!

When trying to solve a system with a condition number on the order of 1E12 or greater on the HP49G+, the traditional methods will not give reliable results; use the SVD method instead.

Edited: 18 May 2006, 5:53 p.m. after one or more responses were posted

Re: How does ill-conditioning happen? typo?

Message #2 Posted by ***Marcus von Cube, Germany*** *on 18 May 2006, 9:05 a.m.,*
in response to message #1 by Rodger Rosenbaum

Quote:

```
[[1.001 2 3]     [[ 1]    [[.001]          [[1.001 2 3]     [[ 1000000]    [[1000]
 [  4   5 6]  *   [ 1]  =  [  0 ]   and     [  4   5 6]  *   [ 1000000]  =  [  0 ]
 [  7   8 9]]     [-1]]    [  0 ]]          [  7   8 9]]     [-1000000]]    [  0 ]]
```

I think you just used the wrong matrix elements. The last colum should probably read 3,9,15.

Marcus

Re: How does ill-conditioning happen? typo?

Message #3 Posted by ***Rodger Rosenbaum*** *on 18 May 2006, 12:19 p.m.,*
in response to message #2 by Marcus von Cube, Germany

It's certainly possible. That post took me hours to put together, and I proofread it many times, finding quite a few errors; I'm sure there are more.

But, could you be more specific about what you mean? Show what you think the correct version should be.

Re: How does ill-conditioning happen? typo?

Message #4 Posted by ***Rodger Rosenbaum*** *on 18 May 2006, 4:21 p.m.,*
in response to message #2 by Marcus von Cube, Germany

OK. I see what the mistake was, and I've edited it. Keep reading and let me know if you find any more errors.

Re: How does ill-conditioning happen? (long)

Message #5 Posted by ***Marcus von Cube, Germany*** *on 18 May 2006, 5:05 p.m.,*
in response to message #1 by Rodger Rosenbaum

Quote:

If we multiply A * [[6.33..., .999...,-4.33...]]T, we get [[13.99..., 15.99..., 13.99...]]T, slightly perturbed from the [[14 15 14]]T we expect.

Another (minor) typo: [[14 15 14]]T should read [[14 16 14]]T

Rodger, I enjoyed your post very much, it helps a lot in following some recent discussions about linear systems and mean matrices.

Marcus

Re: How does ill-conditioning happen? (long)

Message #6 Posted by ***Rodger Rosenbaum*** *on 18 May 2006, 5:55 p.m.,*
in response to message #5 by Marcus von Cube, Germany

Fixed. Thanks.

Re: How does ill-conditioning happen? (long)

Message #7 Posted by ***Marcus von Cube, Germany*** *on 19 May 2006, 2:31 a.m.,*
in response to message #6 by Rodger Rosenbaum

Rodger,

the articles about SVD (including this one) should be migrated to the articles forum. They are worth it.

Marcus

I cannot resist [Q: How does ill-conditioning happen?]

Message #8 Posted by ***Karl Schneider*** *on 20 May 2006, 12:26 a.m.,*
in response to message #1 by Rodger Rosenbaum

Ill-conditioning is a direct consequence of two of the Seven Deadly Sins: Sloth and Gluttony. A regimen of regular exercise and dietary discipline can help prevent "ill-conditioning", which has reached endemic proportions in the US, and is spreading throughout much of the developed world.

-- KS

Edited: 20 May 2006, 12:28 a.m.

Re: I cannot resist [Q: How does ill-conditioning happen?]

Message #9 Posted by ***Rodger Rosenbaum*** *on 20 May 2006, 2:05 a.m.,*
in response to message #8 by Karl Schneider