**Solución de sistemas especiales**

**TABLE OF CONTENTS**

* [Introduction](http://www.fim.utp.ac.pa/Members/fernando.castillo/metodos-numericos/laboratorio-parte-2/laboratorio-7#INTRODUCTION)
* [Tridiagonal Matrices](http://www.fim.utp.ac.pa/Members/fernando.castillo/metodos-numericos/laboratorio-parte-2/laboratorio-7#TRIDIAGONAL_MATRICES)
* [Compact Tridiagonal PLU Factors](http://www.fim.utp.ac.pa/Members/fernando.castillo/metodos-numericos/laboratorio-parte-2/laboratorio-7#COMPACT_TRIDIAGONAL_PLU_FACTORS)
* [Matrix Powers](http://www.fim.utp.ac.pa/Members/fernando.castillo/metodos-numericos/laboratorio-parte-2/laboratorio-7#MATRIX_POWERS)
* [The Jacobi Iteration](http://www.fim.utp.ac.pa/Members/fernando.castillo/metodos-numericos/laboratorio-parte-2/laboratorio-7#THE_JACOBI_ITERATION)
* [Jacobi Iteration Convergence](http://www.fim.utp.ac.pa/Members/fernando.castillo/metodos-numericos/laboratorio-parte-2/laboratorio-7#JACOBI_ITERATION_CONVERGENCE)
* [The Gauss-Seidel Iteration](http://www.fim.utp.ac.pa/Members/fernando.castillo/metodos-numericos/laboratorio-parte-2/laboratorio-7#THE_GAUSS_SEIDEL_ITERATION)
* [ASSIGNMENT](http://www.fim.utp.ac.pa/Members/fernando.castillo/metodos-numericos/laboratorio-parte-2/laboratorio-7#ASSIGNMENT)
* [Answers](http://www.fim.utp.ac.pa/Members/fernando.castillo/metodos-numericos/laboratorio-parte-2/laboratorio-7#ANSWERS)

**Introduction**

We begin our discussion by examining tridiagonal matrices. This is a simple matrix form for which a compact storage method can be developed. We then determine how to compute the PLU factors of such a system, again using compact storage. The reason we are re-examining a problem we've really already solved is that we want to be prepared to solve really large tridiagaonal systems in a way that is efficient in computer memory and operations.

We then consider the results of repeatedly multiplying a vector by the same matrix. If we concentrate simply on the norms of the sequence of results, we see a very simple pattern that can be explained in terms of the norm of the matrix. This result will guide us in our study of the convergence of iterative methods.

There are certain good reasons for not wanting to use Gauss elimination to solve a particular system, particularly if that system is very large (a lot of equations), but "sparse" (only a few nonzero coefficients in each equation). There are other technical details that determine whether this method will actually work or not, but we'll ignore those for now.

**Tridiagonal Matrices**

A tridiagonal matrix has a very simple structure. The only nonzero entries lie along the main diagonal, and the immediate subdiagonal and superdiagonal. A simple example of such a matrix is our second difference matrix, in which the values along the diagonals are actually constant.

It's obvious that a tridiagonal matrix can be stored with much less space than a full matrix. You should realize that it's likely that a tridiagonal linear system can be factored and solved with much less effort as well. If we can properly set up a tridiagonal solver, we can handle problems much too large for MATLAB to deal with using regular storage and regular solution methods.

The first thing to do is to try to get an idea of what we're up against. Let's look at the LU factors and the inverses of several random tridiagonal matrices.

**Experiment**: Copy the files *spline\_tri.m* and *random\_tri.m* from the web page. Using those two files, and the file *dif2.m*, generate a few tridiagonal matrices. Make some experiments so that you can tell me about the pattern of nonzero entries, if any, in:

* the inverse:\_\_\_\_\_\_\_\_\_\_
* the PLU factors\_\_\_\_\_\_\_\_\_\_
* the QR factors\_\_\_\_\_\_\_\_\_\_
* the product of two tridiagonal matrices\_\_\_\_\_\_\_\_\_\_

**Compact Tridiagonal PLU Factors**

It looks as though the nonzero values in the PLU factors of a tridiagonal matrix don't take up any more space than the original values. This leads us to the following thought: if we want to solve really big problems involving a tridiagonal matrix, then we're wasting a lot of space that we set aside for entries away from the three main diagonals. After all, we only need three vectors to store the matrix, and it looks like we only need three vectors to store the LU factors.

If we want to store a tridiagonal matrix compactly, we could simply use three vectors, **A1**, **A2** and **A3** for the subdiagonal, diagonal, and superdiagonal entries. The destination of each entry of the original matrix is suggested by this diagram:

A2(1) A3(1) 0 0 0  
 A1(1) A2(2) A3(2) 0 0  
 0 A1(2) A2(3) A3(3) 0  
 0 0 A1(3) A2(4) A3(4)  
 0 0 0 A1(4) A2(5)

while this diagram suggests how the data from the matrix is stored in the three vectors.

A(2,1) A(1,1) A(1,2)  
 A(3,2) A(2,2) A(2,3)  
 A(4,3) A(3,3) A(3,4)  
 A(5,4) A(4,4) A(4,5)  
 A(5,5)

Of course, we really want to compute the PLU factors of the matrix, and if we're going to receive the matrix in this compact format, we have to rewrite our algorithm to handle this new input, and presumably also to create the LU factors in compact form.

Actually, you may have noticed that in some cases, the upper triangular factor had *three* diagonals of nonzeros. This only happens when pivoting rearranges the original order. We're going to assume from here on that we can dispense with pivoting; either our matrices will be guaranteed to have big diagonal entries at all stages of elimination, or we just won't worry about the effects of small diagonal elements, and we won't expect a zero diagonal value.

With those assumptions, we don't need to compute the **P** factor, the **L** factor will consist of a diagonal of 1's and a single subdiagonal of **N-1** entries, and the **U** factor will have a diagonal of **N** and a superdiagonal of **N-1** entries.

The following diagram suggests how the entries of the matrices L and U will be assigned to vectors L, U and U2:

1 0 0 0 0 U(1) U2(1) 0 0 0  
 L(1) 1 0 0 0 0 U(2) U2(2) 0 0  
 0 L(2) 1 0 0 0 0 U(3) U2(3) 0  
 0 0 L(3) 1 0 0 0 0 U(4) U2(4)  
 0 0 0 L(4) 1 0 0 0 0 U(5)

while this diagram suggests how the data from the matrices is stored in the three vectors:

L(2,1) U(1,1) U(1,2)  
 L(3,2) U(2,2) U(2,3)  
 L(4,3) U(3,3) U(3,4)  
 L(5,4) U(4,4) U(4,5)  
 U(5,5)

So we can store the entire tridiagonal matrix in 3\*N-2 entries. We can store all the PLU factor information in 3\*N-2 entries. Would it be possible to start with the compact matrix information and compute the compact factor information? Of course. We really only need to look at our Gauss elimination procedure and think about the special features of a tridiagonal matrix.

A sketch of the code to compute the LU factors would be:

L=A1, U=A2, U2=A3;  
 for i = 2 : n  
 U(i) = U(i) - U2(i-1) \* L(i-1) / U(i-1);  
 L(i-1) = L(i-1) / U(i-1);  
 end

Copy and examine the M-file *tri\_factor.m* which implements this algorithm.

This code is surprisingly short! You can explain the first assignment very easily. On the first step of Gauss elimination, we take the first diagonal element (U(1)) as our pivot element. Then we have to eliminate the entry below it, which is L(1). To do that, we subtract L(1)/U(1) times the first row from the second, and that causes U(2) to change. The second assignment simply records the multiplier used by the first assignment.

A sketch of the code to solve a factored linear system would be:

y(1) = b(1)  
 for i = 2 : n  
 y(i) = b(i) - L(i-1) \* x(i-1);  
 end  
  
 for i = n : -1 : 2  
 x(i) = y(i) / U(i);  
 y(i-1) = y(i-1) - U2(i-1) \* x(i);  
 end  
 x(1) = y(1) / U(1)

Copy and examine the M-file *tri\_solve.m* which implements this algorithm.

**M File:** - How are we going to know whether our answer is right? We will, because you are going to write an M-file called *tri\_resid.m* which computes the residual for a tridiagonal system stored in the compact format we have discussed. The routine should have the form:

**function r = tri\_resid ( A1, A2, A3, x, b )**

and, of course, it should compute the residual **r=A\*x-b**. This is a really a simple job, but it's hard to program correctly!

**Exercise**: - copy the M-file *tri\_lesp.m*, which defines a Lesp matrix using the compact tridiagonal format. Set up a Lesp matrix of order 5. Define the correct solution of the linear system to be **x=(1,2,3,4,5)**. Now we need to set up the right hand side **b**. Figure out a clever way to use your residual routine to do this for you. (Give up? Look at the end of the lab!) Then factor the matrix using *tri\_factor* and solve it using *tri\_solve*. If all is well, you should get the correct solution.

**Worksheet**:

Define A: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
 Define x: x =\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
 Set b: b =\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
 Factor A: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
 Solve system: x =\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
 Residual: r =\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
 Residual small? \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

(I chose the Lesp matrix because every entry is different. If we had used the difference matrix, all the elements are equal along a diagonal, and you could make many errors in your residual routine without them showing up).

We'll put aside tridiagonal matrices for a moment, and prepare to look at iterative methods. Don't be surprised if we end up seeing more about solving tridiagonal matrices, both directly and iteratively, shortly!

**Matrix Powers**

We want to take a look at iterative methods for linear systems. Before we do, we will first consider a very simple iterative method, involving repeated multiplication of a vector by a matrix. This example will introduce us to iterative methods, remind us of the use of matrix and vector norms, and prepare us for a convergence result we will need later in this lab, and the concept of eigenvectors in the next lab.

Our first experiment starts with two simple tools, a matrix **A** and a vector **x**. We ask the following question:

*What can happen with the sequence of vectors* **(x, A\*x, A2\*x,..., An\*x,...)***?*

Naturally, the answer to this question will depend on our choice of matrix and vector. So let's simplify the question a bit, and ask if we can at least say something about the *norms* of the sequence of vectors. It turns out we can; the sequence of norms behaves almost like successive powers of a given real number, namely, the sequence either blows up, goes to zero, or tends to oscillate in some limited range.

Assuming we are using a compatible matrix norm, then we know

**||A\*x|| <= ||A|| \* ||x||**

We also know that if **||A|| < 1** and **x** nonzero, then **||A\*x||** must be smaller than **||x||**, **||A2\*x||** even smaller, and in fact,

**||A|| < 1 =>   
limit ||An\*x|| < limit ||A||n \* ||x|| = 0**

hence

**||A|| < 1 => limit An \* x = 0**

where the right hand side is actually a zero vector.

**Quiz**: In the above statements, we did not specify the norm used. The only restriction is that the matrix norm used must be compatible with the vector norm. From time to time, we like to characterize a matrix in terms of its spectral norm, that is, the magnitude of the largest eigenvalue of the matrix. While this is a matrix norm, it is not compatible with any vector norm, for somewhat technical reasons. However, we recall that there is always a compatible norm that is "close" to the spectral norm. In detail, if the spectral norm of **A** is **||A||S**, then for any epsilon, there is always some compatible matrix norm **||\*||C** such that

**||A||S <= ||A||C <= ||A||S + epsilon**

Suppose then, that I know one of the following statements about a matrix in terms of the spectral norm. When can I make the same statement, but using a compatible norm?

1. **||A||S < 1**
2. **||A||S <= 1**
3. **||A||S = 1**

(Time's up! Refer to the answers at the end.)

**Exercise**: Consider the following matrices:

* A1 = lesp(5);
* A2 = random\_orthogonal(5);
* A3 = markov(5);
* A4 = jacobi\_dif(5);

(Copy the M-files from the web page if you don't have them already).

**Worksheet**: Compute the "dominant" eigenvalue of each matrix, that is, the eigenvalue of maximum magnitude. You can compute the eigenvalues of a matrix using the command **eig(A)**, and the magnitude by **abs(eig(A))**.

Dominant  
 Matrix eigenvalue   
 A1 \_\_\_\_\_\_\_\_\_\_   
 A2 \_\_\_\_\_\_\_\_\_\_  
 A3 \_\_\_\_\_\_\_\_\_\_  
 A4 \_\_\_\_\_\_\_\_\_\_

**Worksheet**: For each matrix, and the starting vector **x=ones(5,1)**, look for a tendency in the ratio **||An+1\*x|| / ||An\*x||**, using any vector norm you like. What pattern do you see? Interesting results would be "blows up", "oscillates", "tends to 7", or "goes to zero".

ratio tendency   
 A1 \_\_\_\_\_\_\_\_\_\_\_\_\_  
 A2 \_\_\_\_\_\_\_\_\_\_\_\_\_  
 A3 \_\_\_\_\_\_\_\_\_\_\_\_\_  
 A4 \_\_\_\_\_\_\_\_\_\_\_\_\_

Now compare your two worksheets. Does the evidence support the idea that the rate of growth of the matrix-vector products is related to the size of the dominant eigenvalue?

**The Jacobi Iteration**

To carry out an iterative solution of a linear system, we suppose that we already have **x0**, a guess for the answer. We can compute the residual error, which we might symbolize as **e0**:

**e0 = A \* x0 - b**

and, assuming that **||e0||** is probably large, we want to try to compute an "improved" estimate of the solution, **x1**.

In the Jacobi method, we try to come up with an improved estimate by adjusting the first variable so that the first component of the residual goes away; then adjusting the second variable (putting the first variable back to its original value) to make the second residual component zero, and so on.

To see what I mean, suppose we had the following system:

**4 x + 2 y + z = 11 x + 5 y + 2 z = 17 2 x + y + 4 z = 15**

and that our initial guess was

**(x0,y0,z0)** = (1,1,1).

Then we solve the first equation for **x** to get our improved estimate:

**x1 = ( 11 - 2 \* y0 - 1 \* z0 ) / 4 = 8 / 4 = 2**

Then we solve the second equation for **y**:

**y1 = ( 17 - 1 \* x0 - 2 \* z0 ) / 5 = 14 / 5 = 2.8**

and finally, we compute:

**z1 = ( 15 - 2 \* x0 - y0 ) / 4 = 12 / 4 = 3.0**

so that at the end of our first iteration, we have

**(x1,y1,z1) = ( 2, 2.8, 3.0 ).**

Now if we check the residual of this "solution", we discover that it's not zero! That's because we improved each variable, assuming the others would stay fixed. Now it turns out that this procedure can be a reasonable one that converges if the matrix satisfies certain technical requirements.

**M File** - write an M file called *jacobi.m* which takes a single step of Jacobi iteration. The function should have the form:

**function x = jacobi ( A, xold, b )**

Here, **A** is your matrix, **xold** your starting point or previous iterate, and **b** your right hand side.

**Design suggestions** - Your function might have the following form:

**[n,n] = size ( A ) for i = 1 : n x(i) = ? end**

The part with the question mark will take a little work. You may need to add another loop, and an **if** statement, in order to carry out the computation. It might be worthwhile to go back to the example and determine what matrix and vector entries you use on each step.

**Exercise** - Consider the following linear system **A\*x=b** where **A** is the difference matrix. For a given problem size **n=5**, define a true solution **x=[1:n]'**, determine the right hand side **b**. Starting from **x0=zeros(5,1)**, complete the following table:

Step  
 0 [ 0.0, 0.0, 0.0, 0.0, 0.0 ]  
 1 [ 0.0, 0.0, 0.0, 0.0, 3.0 ]  
 2 [ 0.0, 0.0, 0.0, 1.5, 3.0 ]  
 3 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
 4 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
 5 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
 10 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
 20 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Do you think the iteration is converging? Is the error decreasing in some regular way?

Since we're using MATLAB, and solving small problems, it's hard to realize that we could ever want to use an iterative method when direct methods faster and reliable. But in fact, full matrix storage and direct solution methods **are a luxury** that we can only afford for small problems. It is common to have to solve linear systems with 100,000 variables, or even millions. Such problems are often so big that we can't even store the coefficient matrix in the usual way, and Gauss elimination is often much too slow to give us any results. Iterative techniques can be faster and cheaper, especially when the matrix has mostly zero entries. The difference matrix and other tridiagonal systems are examples of the special systems that will be suitable for such treatment.

**Jacobi Iteration Convergence**

When we are dealing with an iterative process, we need to know *whether* it will converge; if we can show that, we then want to know *how fast* this convergence will be. We will briefly consider these questions for the Jacobi iterative method.

To understand the process, we need to write a relationship between succesive iterates. For the Jacobi process, we begin by decomposing our matrix A into lower triangular, diagonal and upper triangular parts:

**A = L + D + U**

Now, for an exact solution **x**, we could write:

**( L + D + U ) \* x = b**

and, if you think about it, the relationship between successive Jacobi iterates is simply a variation of the previous expression:

**D \* xk+1 = b - ( L + U ) \* xk**

(Here, the subscript denotes an iterate index, not a component index). We can rewrite this relationship as:

**xk+1 = D-1 ( b - ( L + U ) \* xk )**

Now, if we write the error as

**ek+1 = xk+1 - x**

and we denote by **J** the *Jacobi iteration matrix*:

**J = D-1 \* ( L + U )**

then with a little work, we have the following:

**ek+1 = J \* ek = ... = Jk e0**

What is the obvious thing to require of the matrix **J**, in order to guarantee the convergence of the Jacobi iteration for any starting point? \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_? This result also gives us a convergence rate for the iteration.

If we have the luxury of MATLAB, we can easily construct and analyze the Jacobi iteration matrix **J** from the system matrix. Here's the MATLAB commands you need:

**D = diag ( diag ( A ) ) L = tril ( A ) - D U = triu ( A ) - D J = inv ( D ) \* ( L + U ) eig ( J )**

Two more typical methods of guaranteeing convergence:

* If **A** is strictly diagonally dominant (for each row, the magnitude of the diagonal entry is strictly larger than the sum of the magnitudes of the off-diagonal elements) (easy to check);
* if **A** is positive definite symmetric (not so easy to check).

**Exercise**: Define the following quantities:

10 3 1 14 0  
 A = 2 -10 3 b = -5 x0 = 0  
 1 3 10 14 0

Will Jacobi iteration converge for this matrix? Why? What is the Jacobi iteration matrix **J**. What is the expected rate of convergence? Starting from x0, compute the first five iterates. The exact solution is [1,1,1]. Determine the infinity norm error in the first five iterates, and ratios between successive errors:

K XK ||E(K)|| ||E(K)|| / ||E(K-1)||  
 0 [ 0 0 0] 1 xxxxxxxxxxxxxxx  
 1 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
 2 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
 3 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
 4 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
 5 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**The Gauss-Seidel Iteration**

The idea of Gauss-Seidel iteration is very similar to the Jacobi iteration, except that as soon as we compute an improved estimate for a solution component, we throw away the old value.

Suppose we had the same starting estimate, and linear system, as in the previous example. Then our first computation would be the same: Then we solve the first equation for **x** to get our improved estimate:

**x1 = ( 11 - 2 \* y0 - 1 \* z0 ) / 4 = 8 / 4 = 2**

But now, when we solve the second equation for **y**, the new value of **x** is used:

**y1 = ( 17 - 1 \* x1 - 2 \* z0 ) / 5 = 13 / 5 = 2.6**

and finally:

**z1 = ( 15 - 2 \* x1 - y1 ) / 4 = 8.4 / 4 = 2.1**

so that at the end of our first iteration, we have

**(x1,y1,z1) = ( 2, 2.6, 2.1 ).**

**M File** - copy your Jacobi code, making an M file called *seidel.m*. The function should have the form:

**function x = seidel ( A, xold, b )**

You only need to make one or two adjustments to modify the Jacobi iteration into a Gauss-Seidel iteration.

To understand the Gauss-Seidel iteration, we again write:

**A = L + D + U**

The relationship between successive iterates is simply a variation of the previous expression:

**( L + D ) \* xk+1 = b - U \* xk**

or:

**xk+1 = ( L + D)-1 ( b - U \* xk )**

which means that the Gauss Seidel iteration matrix GS, which controls the error behavior, is

**GS = - ( L + D)-1 \* U**

**Exercise** - Check that you have correctly programmed the Gauss-Seidel iteration. For the system:

10 3 1 14 0  
 A = 2 -10 3 b = -5 x0 = 0  
 1 3 10 14 0

the next two iterates are:

1.40 1.06  
 x1 = 0.78 x2 = 1.02  
 1.02 0.99

**Assignment**

**Assignment**: As for the previous exercise, define the quantities:

10 3 1 14 0  
 A = 2 -10 3 b = -5 x0 = 0  
 1 3 10 14 0

Answer the following questions:

1. What is the Gauss-Seidel iteration matrix **GS** (two decimals is plenty)?

\_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_  
 GS = \_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_  
 \_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_ \_\_\_\_\_\_\_\_\_\_

1. What is the expected rate of convergence? \_\_\_\_\_\_\_\_\_\_
2. Starting from x0, compute the first five iterates, determine the infinity norm errors, and report the successive error ratios:

E1/E0 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_   
 E2/E1 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
 E3/E2 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
 E4/E3 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
 E5/E4 \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Mail your answers to questions 1, 2 and 3 to me.

**Answers**

 To compute the right hand side of a compact tridiagonal linear system using your residual routine, issue the command:

**b = tri\_resid ( A1, A2, A3, x, zeros(n,1) )**

which simply sets **b = A\*x-0**, which is what we want.

Spectral matrix norm quiz:

1. **||A||S < 1** implies **||A||C < 1** for some compatible norm, because if the spectral norm is strictly less than 1, I can find a compatible norm that is less than the spectral norm plus half that distance to 1;
2. **||A||S <= 1**; I cannot conclude that there is a compatible norm which is less than or equal to 1. It is possible that the spectral norm is exactly 1. Then I only know that there are compatible norms that can be made as close to 1 as desired, but I can't guarantee them to be equal to 1.
3. **||A||S = 1**; for the same reason, I cannot conclude that there is a compatible norm with this property.

The point of this exercise is that when you see a statement that something is true whenever the (compatible) matrix norm is *strictly less than* some limit, you can use the spectral norm instead.