Bonus Project

Due: 06.02.2022

Vector Norms

A *vector norm* assigns a size to a vector, in such a way that scalar multiples do what we expect, and the triangle inequality is satisfied. There are four common vector norms in n dimensions:

- The L^1 vector norm

$$\|x\|_1 = \sum_{i=1}^{n} |x_i|$$

- The L^2 (or ``Euclidean") vector norm

$$\|x\|_2 = \sqrt{\sum_{i=1}^{n} |x_i|^2}$$

- The L^p vector norm

$$\|x\|_p = \left(\sum_{i=1}^{n} |x_i|^p \right)^{1/p}$$

- The L^∞ vector norm

$$\|x\|_\infty = \max_{i=1,\ldots,n} |x_i|$$

To compute the norm of a vector x in Matlab:

- $\|x\|_1 =$ `norm(x,1);`
- $\|x\|_2 =$ `norm(x,2)` = `norm(x);`

- $\|x\|_p =$ `norm(x,p);`
- $\|x\|_\infty =$ `norm(x,inf)`

(Recall that `inf` is the Matlab name corresponding to ∞.)

Exercise 1: For each of the following column vectors:

```
x1 = [ 4; 6; 7 ]
x2 = [ 7; 5; 6 ]
x3 = [ 1; 5; 4 ]
```

compute the vector norms, using the appropriate Matlab commands. Be sure your answers are reasonable.

	L1	L2	L Infinity
x1	_____	_____	_____
x2	_____	_____	_____
x3	_____	_____	_____

Matrix Norms

A *matrix norm* assigns a size to a matrix, again, in such a way that scalar multiples do what we expect, and the triangle inequality is satisfied. However, what's more important is that we want to be able to mix matrix and vector norms in various computations. So we are going to be very interested in whether a matrix norm is *compatible* with a particular vector norm, that is, when it is safe to say:

$$\|Ax\| \leq \|A\| \ \|x\|$$

There are four common matrix norms and one ``almost" norm:

- The L^1 or ``max column sum" matrix norm:

$$\|A\|_1 = \max_{j=1,\ldots,n} \sum_{i=1}^{n} |A_{i,j}|$$

  Remark: This is not the same as the L^1 norm of the vector of dimension n^2 whose components are the same as $A_{i,j}$.

- The L^2 matrix norm:

$$\|A\|_2 = \max_{j=1,\ldots,n} \sqrt{\lambda_i}$$

  where λ_i is a (necessarily real and non-negative) eigenvalue of $A^H A$ or

$$\|A\|_2 = \max_{j=1,\ldots,n} \mu_i$$

where μ_i is a singular value of A;

- The L^∞ or ``max row sum'' matrix norm:

$$\|A\|_\infty = \max_{i=1,\ldots,n} \sum_{j=1}^{n} |A_{i,j}|$$

Remark: This is not the same as the L^∞ norm of the vector of dimension n^2 whose components are the same as $A_{i,j}$.

- There is no L^p matrix norm in Matlab.
- The ``Frobenius'' matrix norm:

$$\|A\|_{\mathtt{fro}} = \sqrt{\sum_{i,j=1,\ldots,n} |A_{i,j}|^2}$$

Remark: This is the same as the L^2 norm of the vector of dimension n^2 whose components are the same as $A_{i,j}$.

- The spectral radius (not a norm):

$$\rho(A) = \max |\lambda_i|$$

(only defined for a square matrix), where λ_i is a (possibly complex) eigenvalue of A.

To compute the norm of a matrix A in Matlab:

- $\|A\|_1 =$ `norm(A,1);`
- $\|A\|_2 =$ `norm(A,2)=norm(A);`
- $\|A\|_\infty =$ `norm(A,inf);`
- $\|A\|_{\mathtt{fro}} =$ `norm(A,'fro')`
- See below for computation of $\rho(A)$ (the spectral radius of A)

Compatible Matrix Norms

A matrix can be identified with a linear operator, and the norm of a linear operator is usually defined in the following way.

$$\|A\| = \max_{x \neq 0} \frac{\|Ax\|}{\|x\|}$$

(It would be more precise to use $\sup$ rather than $\max$ here but the surface of a sphere in finite-dimensional space is a compact set, so the supremum is attained, and the maximum is correct.) A matrix norm defined in this way is said to be ``vector-bound" to the given vector norm.

In order for a matrix norm to be consistent with the linear operator norm, you need to be able to say the following:

$$\|Ax\| \leq \|A\| \ \|x\| \tag{1}$$

but this expression *is not necessarily true* for an arbitrarily chosen pair of matrix and vector norms. When it is true, then the two are ``compatible".

If a matrix norm is vector-bound to a particular vector norm, then the two norms are guaranteed to be compatible. Thus, for any vector norm, there is always at least one matrix norm that we can use. But that vector-bound matrix norm is not always the only choice. In particular, the L^2 matrix norm is difficult (time-consuming) to compute, but there is a simple alternative.

Note that:

- The L^1, L^2 and L^∞ matrix norms can be shown to be vector-bound to the corresponding vector norms and hence are guaranteed to be compatible with them;
- The Frobenius matrix norm is not vector-bound to the L^2 vector norm, but is compatible with it; the Frobenius norm is much faster to compute than the L^2 matrix norm (see Exercise 6 below).

  Exercise 2: Consider each of the following column vectors:

```
x1 = [ 4; 6; 7 ];
x2 = [ 7; 5; 6 ];
x3 = [ 1; 5; 4 ];
```

  and each of the following matrices

```
A1 = [38   37   80
      53   49   49
```

```
       23   85   46];

A2 = [77   89   78
       6   34   10
      65   36   26];
```

Check that the compatibility condition (1) holds for each case in the following table by computing the ratios

$$r_{p,q}(A,x) = \frac{\|Ax\|_q}{\|A\|_p \|x\|_q}$$

and filling in the following table. The third column should contain the letter ``S" if the compatibility condition is satisfied and the letter ``U" if it is unsatisfied.

Suggestion: I recommend you write a script m-file for this exercise. If you do, please include it with your summary.

norm (p)	Matrix norm (q)	Vector S/U	r(A1,x1)	r(A1,x2)	r(A1,x3)	r(A2,x1)	r(A2,x2)	r(A2,x3)
1	1	___	_____	_____	_____	_____	_____	_____
1	2	___	_____	_____	_____	_____	_____	_____
1	inf	___	_____	_____	_____	_____	_____	_____
2	1	___	_____	_____	_____	_____	_____	_____
2	2	___	_____	_____	_____	_____	_____	_____
2	inf	___	_____	_____	_____	_____	_____	_____
inf	1	___	_____	_____	_____	_____	_____	_____
inf	2	___	_____	_____	_____	_____	_____	_____
inf	inf	___	_____	_____	_____	_____	_____	_____

Types of Errors

A natural assumption to make is that the term ``error" refers always to the difference between the computed and exact ``answers." We are going to have to discuss several kinds of error, so let's refer to this first error as ``solution error" or ``forward error." Suppose we want to solve a linear system of the form $A\mathbf{x} = \mathbf{b}$, (with exact solution $\mathbf{x}$) and we computed $\mathbf{x}_{\text{approx}}$. We define the solution error as $\|\mathbf{x}_{\text{approx}} - \mathbf{x}\|$.

Usually the solution error cannot be computed, and we would like a substitute whose *behavior* is acceptably close to that of the solution error. One example would be during an iterative solution process where we would like to monitor the progress of the iteration but we do not yet have the solution and cannot compute the solution error. In this case, we are interested in the ``residual error" or ``backward error," which is defined by $\|A\mathbf{x}_{\text{approx}} - \mathbf{b}\| = \|\mathbf{b}_{\text{approx}} - \mathbf{b}\|$ where, for convenience, we have defined the variable $\mathbf{b}_{\text{approx}}$ to equal $A\mathbf{x}_{\text{approx}}$. Another way of looking at the residual error is to see that it's telling us the difference between the right hand side that would ``work" for $\mathbf{x}_{\text{approx}}$ versus the right hand side we have.

If we think of the right hand side as being a target, and our solution procedure as determining how we should aim an arrow so that we hit this target, then

- The solution error is telling us how badly we aimed our arrow;
- The residual error is telling us how much we would have to move the target in order for our badly aimed arrow to be a bullseye.

There are problems for which the solution error is huge and the residual error tiny, and all the other possible combinations also occur.

Exercise 3: For each of the following cases, compute the Euclidean norm (two norm) of the solution error and residual error and characterize each as ``large" or ``small." (For the purpose of this exercise, take ``large" to mean greater than 1 and ``small" to mean smaller than 0.01.) In each case, you must find the true solution first (Pay attention! In each case you can solve in your head for the true solution, `xTrue`.), then compare it with the approximate solution `xApprox`.

1. `A=[1,1;1,(1-1.e-12)],b=[0;0],xApprox=[1;-1]`
2. `A=[1,1;1,(1-1.e-12)],b=[1;1],xApprox=[1.00001;0]`
3. `A=[1,1;1,(1-1.e-12)],b=[1;1],xApprox=[100;100]`
4. `A=[1.e+12,-1.e+12;1,1],b=[0;2],xApprox=[1.001;1]`

Case	Residual	large/small	xTrue	Error	large/small
1	_____	_____	_____	_____	_____
2	_____	_____	_____	_____	_____
3	_____	_____	_____	_____	_____
4	_____	_____	_____	_____	_____

The linear system ``problem"

The linear system ``problem'' can be posed in the following way. Find an n -vector $\mathbf{x}$ that satisfies the matrix equation

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{1}$$

where $\mathbf{A}$ is a $m \times n$ matrix and $\mathbf{b}$ is a m-vector. In Matlab, both $\mathbf{x}$ and $\mathbf{b}$ are column vectors.

You probably already know that there is ``usually'' a solution if the matrix is square, (that is, if $m = n$). We will concentrate on this case for now. But you may wonder if there is an intelligent response to this problem for the cases of a square but singular matrix, or a rectangular system, whether overdetermined or underdetermined.

At one time or another, you have probably been introduced to several algorithms for producing a solution to the linear system problem, including Cramer's rule (using determinants), constructing the inverse matrix, Gauß-Jordan elimination and Gauß factorization. We will see that it is usually not a good idea to construct the inverse matrix and we will focus on Gauß factorization for solution of linear systems.

We will be concerned about several topics:

> **Efficiency**: what algorithms produce a result with less work?
> **Accuracy**: what algorithms produce an answer that is likely to be more accurate?
> **Difficulty**: what makes a problem difficult or impossible to solve?
> **Special cases**: how do we solve problems that are big? symmetric? banded? singular? rectangular?

The inverse matrix

A classic result from linear algebra is the following:
Theorem *The linear system problem (1) is uniquely solvable for arbitrary* $\mathbf{b}$ *if and only if the inverse matrix* $\mathbf{A}^{-1}$ *exists. In this case the solution* $\mathbf{x}$ *can be expressed as* $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$.

So what's the catch? There are a few:

- Computing the inverse takes a lot of time-more time than is necessary if you only want to know the solution of a particular system.

- In cases when the given matrix actually has very many zero entries, as is the case with the `dif2` matrix, computing the inverse is enormously more difficult and time consuming than merely solving the system. And it is quite often true that the inverse matrix requires far more storage than the matrix itself. The second difference matrix, for example, is an ``M-matrix" and it can be proved that all the entries in its inverse are positive numbers. You only need about $3n$ numbers to store the `dif2` matrix because you can often avoid storing the zero entries, but you need n^2 numbers to store its inverse.
- Sometimes the inverse is so inaccurate that it is not worth the trouble to multiply by the inverse to get the solution.

In the following exercises, you will see that constructing the inverse matrix takes time proportional to n^3 (for an $n \times n$ matrix), as does simply solving the system, but that solving the system is several times faster than constructing the inverse matrix. You will also see that matrices with special formats, such as tridiagonal matrices, can be solved very efficiently. And you will see even simple matrices can be difficult to numerically invert.

You will be measuring elapsed time in order to see how long these calculations take. The problem sizes are designed to take a modest amount of time (less than 6 minutes) on newer computers.

WARNING: Most newer computers have more than one processor (core). By default, Matlab will use all the processors available by assigning one ``thread" to each available processor. This procedure will mess up our timings, so you should tell Matlab to use only a single thread. To do this, you should start up Matlab with the command line

```
matlab -singleCompThread
```

If you do not know how to start up Matlab from the command line, use the following command at the beginning of your session.

```
maxNumCompThreads(1);
```

Matlab will warn you that this command will disappear in the future, but it should work fine now.

Exercise 4: The Matlab command `inv(A)` computes an approximation for the inverse of a matrix `A`. Do not worry for now just how it does it, but be assured that it uses one of the most efficient and reliable methods available.

Matlab provides the commands `tic` and `toc` to measure computational time. The `tic` command starts the stopwatch, and the `toc` command stops it, either printing the elapsed time or returning its value as in the expression `elapsedTime=toc;`. The times are in seconds.
(**Note:** `tic` and `toc` measure *elapsed* time. When a computer is doing more than one thing, the `cputime` function can be used to measure how much computer time is being used for this task alone.)

1. Copy the following code to a Matlab function m-file named `exer4.m` and modify it to produce information for the table below. Be sure to add comments and your name and the date. Include the file with your summary.

```
function elapsedTime=exer4(n)
% elapsedTime=exer4(n)
% comments

% your name and the date

if mod(n,2)==0
  error('Please use only odd values for n');
   end

   A = magic(n);    % only odd n yield invertible matrices
   b = ones(n,1);   % the right side vector doesn't change the
time
   tic;
   Ainv =  ???      % compute the inverse matrix
   xSolution = ???  % compute the solution
   elapsedTime=toc;
```

18. You have seen in lecture that the time required to invert an $n \times n$ matrix should be proportional to n^3. Fill in the following table, where the column entitled ``ratio" should contain the ratio of the time for n divided by the time for the preceeding value of n.
    (**Note:** timings are not precise! Furthermore, the first time a function is used results in Matlab reading the m-file and ``compiling it," and that takes considerable time. The last row of this table may take several minutes!)
    Remark: Compute the first line of the table twice and use the second value. The first time a function is called involves substantial overhead that later calls do not require.

```
     Time to compute inverse matrices
      n     time      ratio
     161  _________
     321  _________  _________
     641  _________  _________
    1281  _________  _________
    2561  _________  _________
    5121  _________  _________
   10241  _________  _________
```

28. Are these solution times roughly proportional to n^3?

Exercise 5: Matlab provides a special operator, the backslash (\) operator, that is designed to solve a linear system without computing the inverse. It is used in the following way, for a matrix `A` and column vector `b`.

```
x=A\b;
```

It may help you to remember this operator if you think of the `A` as being ``underneath" the slash. The effect of this operator is to find the solution of the system of equations `A*x=b`.

The backslash looks strange, but there is a method to this madness. You might wonder why you can't just write `x=b/A`. This would put the column vector `b`to the left of the matrix `A` and that is the wrong order of operations.

1. Copy `exer4.m` to `exer5.m` and replace the inverse matrix computation and solution with an expression using the Matlab ``\'' command, and fill in the following table

```
  Time to compute solutions
    n     time      ratio
  161  _________
  321  _________  _________
  641  _________  _________
 1281  _________  _________
 2561  _________  _________
 5121  _________  _________
10241  _________  _________
```

11. Are these solution times roughly proportional to n^3 ?
12. Compare the times for the inverse and solution and fill in the following table

```
     Comparison of times
      n   (time for inverse)/(time for solution)
     161              _________
     321              _________
     641              _________
    1281              _________
    2561              _________
    5121              _________
   10241              _________
```

22. Theory shows that computation of a matrix inverse should take approximately three times as long as computation of the solution. Are your results consistent with theory?

You should be getting the message: ``**You should never compute an inverse when all you want to do is solve a system.**''

Warning: the ``\'' symbol in Matlab will work when the matrix `A` is *not square.* In fact, sometimes it will work when `A` actually is a vector. The results are not usually what you expect and no error message is given. Be careful of this potential for error when using ``\''. A similar warning is true for the `/` (division) symbol because Matlab will try to ``interpret'' it if you use it with matrices or vectors and you will get ``answers'' that are not what you intend.

Gauß factorization

The standard method for computing the inverse of a matrix is Gaußian elimination. You already know this algorithm, perhaps by another name such as row reduction..

First, we will prefer to think of the process as having two separable parts, the ``factorization" or ``forward substitution" and ``back substitution" steps.

We now turn to writing code for the Gauß factorization. The discussion in this lab assumes that you are familiar with Gaußian elimination (sometimes called ``row reduction"), and is entirely self-contained. Because, however, interchanging rows is a source of confusion, we will assume at first that no row interchanges are necessary.

The idea is that we are going to use the row reduction operations to turn a given matrix $\mathbf{A}$ into an upper triangular matrix (all of whose elements *below* the main diagonal are zero) $\mathbf{U}$ and at the same time keep track of the multipliers in the lower triangular matrix $\mathbf{L}$. These matrices will be built in such a way that $\mathbf{A}$ could be reconstructed from the product $\mathbf{LU}$ at any time during the factorization procedure.

Alert: The fact that $\mathbf{A} = \mathbf{LU}$ even before the computation is completed can be a powerful debugging technique. If you get the wrong answer at the end of a computation, you can test each step of the computation to see where it has gone wrong. Usually you will find an error in the first step, where the error is easiest to understand and fix.

Let's do a simple example first. Consider the matrix

$$\mathbf{A} = \begin{pmatrix} 2 & 4 \\ 1 & 9 \end{pmatrix}.$$

There is only one step to perform in the row reduction process: turn the 1 in the lower left into a 0 by subtracting half the first row from the second. Convince yourself that this process can be written as

$$\begin{pmatrix} 1 & 0 \\ -\frac{1}{2} & 1 \end{pmatrix} \begin{pmatrix} 2 & 4 \\ 1 & 9 \end{pmatrix} = \begin{pmatrix} 2 & 4 \\ 0 & 7 \end{pmatrix}. \tag{2}$$

If you want to write, $\mathbf{A} = \mathbf{LU}$, you need to have $\mathbf{L}$ be the inverse of the first matrix on the left in (2). Thus

$$\mathbf{L} = \begin{pmatrix} 1 & 0 \\ \frac{1}{2} & 1 \end{pmatrix}$$

and

$$\mathbf{U} = \begin{pmatrix} 2 & 4 \\ 0 & 7 \end{pmatrix}$$

are matrices that describe the row reduction step ($\mathbf{L}$) and its result ($\mathbf{U}$) in such a way that the original matrix $\mathbf{A}$ can be recovered as $\mathbf{A} = \mathbf{LU}$.

The Hilbert matrix

The Hilbert matrix is related to the interpolation problem on the interval [0,1]. The matrix is given by the formula $\mathbf{A}_{i,j} = 1/(i+j-1)$. For example, with $n = 5$ the Hilbert matrix is:

$$\begin{pmatrix} \frac{1}{1} & \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} \\ \frac{1}{2} & \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} \\ \frac{1}{3} & \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} \\ \frac{1}{4} & \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} \\ \frac{1}{5} & \frac{1}{6} & \frac{1}{7} & \frac{1}{8} & \frac{1}{9} \end{pmatrix}$$

The Hilbert matrix arises in interpolation and approximation contexts because it happens that $\mathbf{A}_{i,j} = \int_0^1 (x^{i-1})(x^{j-1})dx$. The Hilbert matrix is at once nice because its inverse has integer elements and also not nice because it is difficult to compute the inverse accurately using the usual formulæ to invert a matrix.

Now, suppose you have a 5 by 5 Hilbert matrix (switching to Matlab notation) `A`, and you wish to perform row reduction steps to turn all the entries in the first column below the first row into zero, and keep track of the operations in the matrix `L` and the result in `U`. Convince yourself that the following code does the trick.

Matlab has special functions for the Hilbert matrix and its inverse, called `hilb(n)` and `invhilb(n)`, and we will be using these functions in this lab.

```
n=5;
Jcol=1;
A=hilb(5);
L=eye(n);     % square n by n identity matrix
U=A;
for Irow=Jcol+1:n
  % compute Irow multiplier and save in L(Irow,Jcol)
  L(Irow,Jcol)=U(Irow,Jcol)/U(Jcol,Jcol);

  % multiply row "Jcol" by L(Irow,Jcol) and subtract from row "Irow"
  % This vector statement could be replaced with a loop
  U(Irow,Jcol:n)=U(Irow,Jcol:n)-L(Irow,Jcol)*U(Jcol,Jcol:n);
end
```

Exercise 6:

1. Use the above code to compute the first row reduction steps for the matrix `A=hilb(5)`. Print the resulting matrix `U` and include it in the summary report. Check that all entries in the first column in the second through last rows are zero or roundoff. (If any are non-zero, you have an error somewhere. Find the error before proceeding.)
2. Multiply `L*U` and confirm for yourself that it equals `A`. An easy way to do this is to compute `norm(L*U-A,'fro')/norm(A,'fro')`. (As you have seen, the Frobenius norm is faster to compute than the default 2-norm, especially for large matrices.)
3. Use the code given above as a model for an m-file called `gauss_lu.m` that performs Gaußian reduction without pivoting. The function should start out

```
function [L,U]=gauss_lu(A)
% function [L,U]=gauss_lu(A)
% performs an LU factorization of the matrix A using
% Gaussian reduction.
% A is the matrix to be factored.
% L is a lower triangular factor with 1's on the diagonal
   % U is an upper triangular factor.
   % A = L * U

   % your name and the date
```

14. Use your routine to find `L` and `U` for the Hilbert matrix of order 5. Include `L` and `U` in your summary.
15. What is `U(5,5)`, to at least four significant figures?
16. Verify that `L` is lower triangular (has zeros above the diagonal) and `U` is upper triangular (has zeros below the diagonal).
17. Confirm that `L*U` recovers the original matrix.
18. The Matlab expression `R=rand(100,100)` will generate a 100×100 matrix of random entries. Do not print this matrix-it has 10,000 numbers in it.Use `gauss_lu` to find its factors `LR` and `UR`.
    - Use norms to confirm that `LR*UR=R`, without printing the matrices.
    - Use `tril` and `triu` to confirm that `LR` is lower triangular and `UR` is upper triangular, without printing the matrices.

It turns out that omitting pivoting leaves the function you just wrote vulnerable to failure.

Exercise 7:

1. Compute `L1` and `U1` for the matrix

```
A1 = [ -2   1   0   0   0
        1   0   1  -2   0
        0   0   0   1  -2
        1  -2   1   0   0
        0   1  -2   1   0]
```

   The method should fail for this matrix, producing matrices with infinity (`inf`) and ``Not a Number'' (`NaN`) in them.

7. On which step of the decomposition (values of `Irow` and `Jcol`) does the method fail? Why does it fail?
8. What is the determinant of `A1`? What is the condition number (`cond(A1)`)? Is the matrix singular or ill-conditioned?

In Gaußian elimination it is entirely possible to end up with a zero on the diagonal, as the previous exercise demonstrates. Since you cannot divide by zero, the procedure fails. It is also possible to end up with small numbers on the diagonal and big numbers off the diagonal. In this case, the algorithm doesn't fail, but roundoff errors can overwhelm the procedure and result in wrong answers. One solution to these difficulties is to switch rows and columns around so that the largest remaining entry in the matrix ends up on the diagonal. Instead of this ``full pivoting'' strategy, it is almost as effective to switch the rows only, instead of both rows and columns. This is called ``partial pivoting..