Lab 5: Norms for Measuring Size
MATH2071, University of Pittsburgh, Spring 2023

1 Introduction

The objects we work with in linear systems are vectors and matrices. In order to make statements about the size of these objects, and the errors we make in solutions, we want to be able to describe the "sizes" of vectors and matrices, which we do by using *norms*.

We then need to consider whether we can bound the size of the product of a matrix and vector, given that we know the "size" of the two factors. In order for this to happen, we will need to use matrix and vector norms that are *compatible*. These kinds of bounds will become very important in error analysis.

We will then consider the notions of *forward error* and *backward error* in a linear algebra computation.

From the definitions of norms and errors, we can define the *condition number* of a matrix, which will give us an objective way of measuring how "bad" a matrix is, and how many digits of accuracy we can expect when solving a particular linear system.

We will then look at one useful matrix example: a tridiagonal matrix with well-known eigenvalues and eigenvectors.

This lab will take two sessions.

2 A matrix gallery for testing

To explore concepts in linear algebra, it is useful to become familiar with a few matrices which can be called by name, and created for any reasonable dimension n. For our work, there is a file called `gallery.py`, which you should download from the web site now. This file defines a number of matrices, including:

- `A = dif2 ( n )`, the second different matrix;
- `A = frank ( n )`
- `A = helmert ( n )`
- `A = jordan_block ( n, alpha )`
- `A = moler3 ( n )`

For `n = 5`, the second difference matrix is:

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

An example of the Frank matrix:

```
A = [ 5 4 3 2 1
      4 4 3 2 1
      0 3 3 2 1
      0 0 2 2 1
      0 0 0 1 1 ]
```

A (rounded) example of the Helmert matrix:

```
A = [ 0.447  0.447  0.447  0.447  0.447
      0.707 -0.707  0.     0.     0.
      0.408  0.408 -0.816  0.     0.
```

```
    0.288  0.288  0.288 -0.866  0.
    0.223  0.223  0.223  0.223 -0.894 ]
```

For `alpha = 7` an example of the Jordan block matrix is:

```
A = [7  1  0  0  0
     0  7  1  0  0
     0  0  7  1  0
     0  0  0  7  1
     0  0  0  0  7 ]
```

An example of the Moler3 matrix:

```
A = [ 1 -1 -1 -1 -1
     -1  2  0  0  0
     -1  0  3  1  1
     -1  0  1  4  2
     -1  0  1  2  5]
```

3 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
$$\|x\|_\infty = \max_{i=1,\dots,n} |x_i|$$

To compute the norm of a vector x:

- $\|x\|_1 =$ `np.linalg.norm(x,1)`;
- $\|x\|_2 =$ `np.linalg.norm(x,2)` $=$ `nnp.linalg.norm(x)`;
- $\|x\|_p =$ `np.linalg.norm(x,p)`;
- $\|x\|_\infty =$ `np.linalg.norm(x,np.inf)`

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

4 Exercise 1

- Create a file `exercise1.py` for this experiment.
- Define the following three vectors:

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

- For each vector, compute and print the L1, L2 and L infinity norms.

5 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:

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

- 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, equivalently,

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

  where μ_i is a singular value of A;

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

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

- The "Frobenius" matrix norm:

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

- 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:

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

6 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.
- The spectral radius is not really a norm and is not vector-bound to any vector norm, but it "almost" is. It is useful because we often want to think about the behavior of a matrix as being determined by its largest eigenvalue, and it often is. But there is no vector norm for which it is always true that

  $$\|Ax\| \leq \rho(A)\|x\|.$$

  There is a theorem, though, that states that, for any matrix A, and for any chosen value of $\epsilon > 0$, a norm can be found so that $\|Ax\| \leq (\rho(A) + \epsilon)\|x\|$. Note that the norm depends both on ϵ and A.

7 Checking compatibility

Consider the following vectors:

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

and the following matrices

```
A1 = [38   37   80
      53   49   49
      23   85   46]

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

Python computes matrix norms as well, again using `np.linalg.norm()`. In addition, the Frobenius norm of a matrix `A` can be computed by `Anorm = np.linalg.norm ( A, 'fro' )`.

To compute a matrix-vector product $A * x$, you can use a command like `Ax = np.dot ( A, x )`. In mathematics, x is required to be a column vector. The Python command allows x to be stored as a row vector, but produces a result as though it were a column vector.

If we need to, however, we can always insist on using the column vector format. For instance, we could define `x1 = np.array ( [ [4], [6], [7] ] )`.

Now we want to make a simple check of the compatibility of the various combinations of matrix and vector norm. We will do this by computing the ratio

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

using our example vectors and matrices. If, for a given p and q, the ratio is ever greater than 1, then this proves that the matrix norm is not compatible with that vector norm. On the other hand, if all the ratios are less than 1.0, this may suggest compatibility, but is not a proof.

8 Exercise 2

- Create a file `exercise2.py` for this experiment.
- For p = 1, 2, np.inf, 'fro':
- For q = 1, 2, np.inf
- For A = A1, A2
- For x = x1, x2, x3
- Compute r = $||Ax||_q/||A||_p/||x||_q$, and print p, q, r

This will be a very painful and tedious computation unless you take advantage of Python's ability to write nifty `for` loops that exactly correspond to what you want to do, something like this:

```
for p in [ 1, 2, np.inf, 'fro' ]:
  for q in [ 1, 2, np.inf ]:
    for A in [ A1, A2 ]:
      for x in [ x1, x2, x3 ]:
        Compute r, print p, q, r
```

From your results, which pairs (p,q) represent compatible matrix/vector norms?

9 The Spectral Radius

In the above text there is a statement that the spectral radius is not vector-bound with any norm, but that it "almost" is. In this section we will see by example what this means.

In the first place, let's try to see why it isn't vector-bound to the L^2 norm. Consider the following false "proof". Given a matrix A, for any vector $\mathbf{x}$, break it into a sum of orthonormal eigenvectors of A, as:

$$\mathbf{x} = \sum x_i \mathbf{e}_i$$

Then, for λ_i the eigenvalues of A,

$$\begin{aligned}
\|A\mathbf{x}\|_2^2 &= \|\sum \lambda_i x_i \mathbf{e}_i\|_2^2 \\
&= \sum |\lambda_i|^2 |x_i|^2 ||e_i||^2 \quad \text{(by orthogonality of } e_i\text{)} \\
&\leq \max_i |\lambda_i|^2 \sum |x_i|^2 ||e_i||^2 \quad \text{(factor out max eigenvalue)} \\
&\leq \max_i |\lambda_i|^2 \sum |x_i|^2 \quad \text{(by normality of } e_i\text{)} \\
&\leq \rho(A)^2 \|\mathbf{x}\|^2
\end{aligned}$$

and taking square roots completes the "false proof."

Why is this "proof" false? It works if we really can find a complete set of orthonormal eigenvectors for A, which is possible, for example, if A is symmetric and positive definite. In general, however, that is not possible. There may not be a complete set of eigenvectors, and even if so, they might not be normal to each other.

In the next exercise, we will show an example of how the spectral radius fails to be compatible with the L2 norm, using a particular matrix `A` and vector `x`:

Let `A = jordan_block ( 7, 0.5 )`, as defined in the `gallery.py` function.

Let ρ be the spectral radius of A.

Let x be the vector [1,1,1,1,1,1,1].

Define r = $||Ax||_2/||A||_2/||x||_2$ and s = $||Ax||_q/\rho/||x||_q$.

We will now show that $r \leq 1$, as it must be, and we will test whether $s \leq 1$, as it should be if the spectral radius is compatible with the L^2 vector norm.

10 Exercise 3

- Create a file `exercise3.py` for this experiment.
- Define the matrix `A` and vector `x` as listed above.
- Using the command `w, v = np.linalg.eig(A)`, get the eigenvalues `w` and eigenvectors `v` of `A`;
- Determine the value of `rho` = $\rho(A)$, the size of the largest eigenvalue of A.
- Compute and print `r` = $||Ax||_2/||A||_2/||x||_2$;
- Compute and print `s` = $||Ax||_2/rho/||x||_2$;

Conclusion: The spectral radius matrix norm is/is not compatible with the L^2 vector norm?

11 Exercise 4

The spectral radius of the matrix `A` in the previous example is less than 1. It can be shown that this fact guarantees that, for any vector norm, $||A^k x|| \to 0$ as $k \to \infty$; it does not guarantee that this sequence of norms will be strictly decreasing. Let us examine what happens if we repeatedly multiply our example vector x by our example matrix A.

1. Create a file `exercise4.py` for this experiment.
2. Set up the same matrix `A` and vector `x` as in the previous exercise.
3. For `k` from 0 to 40, compute and print $||A^k x||_2$.
4. Plot the values $(k, ||A^k x||_2)$;

12 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.

The norms of the matrix and its inverse exert some limits on the relationship between the forward and backward errors. Assuming we have compatible norms:

$$\|x_{\text{approx}} - x\| = \|A^{-1}A(x_{\text{approx}} - x)\| \le (\|A^{-1}\|)(\|b_{\text{approx}} - b\|)$$

and

$$\|Ax_{\text{approx}} - b\| = \|Ax_{\text{approx}} - Ax\| \le (\|A\|)(\|x_{\text{approx}} - x\|)$$

Put another way,

$$\begin{aligned}\text{(solution error)} &\le \|A^{-1}\|\text{(residual error)}\\ \text{(residual error)} &\le \|A\|\text{(solution error)}\end{aligned}$$

13 Exercise 5

Consider the following four examples of a matrix `A`, right hand side `b`, approximate solution `x`, and exact solution `t`:

1. `A1=[[1,1],[1,(1-1.e-12)]]`, `b1=[0,0]`, `x1=[1,-1]`, `t=[0,0]`;
2. `A2=[[1,1],[1,(1-1.e-12)]]`, `b2=[1,1]`, `x2=[1.00001,0]`, `t=[1,0]`;
3. `A3=[[1,1],[1,(1-1.e-12)]]`, `b3=[1,1]`, `x3=[100,100]`, `t=[1,0]`;
4. `A4=[[1.e+12,-1.e+12],[1,1]]`, `b4=[0,2]`, `x4=[1.001,1]`, `t=[1,1]`;

1. Create a file `exercise5.py` for this experiment.
2. For each case `A, b, x, t`, compute the residual `r = A*x-b`;
3. For each case `A, b, x, t`, compute the error `e = t - x`;
4. Print the case index, the L^2 norm of `r`, the L^2 norm of `e`;

This is another computation which you can simplify by taking advantage of the Python `for` loop. Instead of the for loop returning just the current loop index (one item), it can return a set of things, as long as you give it a set of sets to choose from. In other words, think about writing something like this:

```
  for k, A, b, x, t in [ [ 1, A1, b1, x1, t1 ], \
                         [ 2, A2, b2, x2, t2 ], \
                         [ 3, A3, b3, x3, t3 ], \
                         [ 4, A4, b4, x4, t4 ] ]:
    Now do your computations with k, A, b, x, t
```

14 Relative error

It's often useful to consider the size of an error relative to the true quantity. If the true solution is x and we computed $x_{\text{approx}} = x + \Delta x$, the relative solution error is defined as

$$\begin{aligned}\text{(relative solution error)} &= \frac{\|x_{\text{approx}} - x\|}{\|x\|} \\ &= \frac{\|\Delta x\|}{\|x\|}\end{aligned}$$

Given the computed solution x_{approx}, we know that it satifies the equation $Ax_{\text{approx}} = b_{\text{approx}}$. If we write $b_{\text{approx}} = b + \Delta b$, then we can define the relative residual error as:

$$\begin{aligned}\text{(relative residual error)} &= \frac{\|b_{\text{approx}} - b\|}{\|b\|} \\ &= \frac{\|\Delta b\|}{\|b\|}\end{aligned}$$

These quantities depend on the vector norm used, they cannot be defined in cases where the divisor is zero, and they are problematic when the divisor is small.

15 sine_bvp.py

Consider the boundary value problem (BVP) for the ordinary differential equation

$$\begin{aligned}u'' &= -\frac{\pi^2}{100}\sin(\frac{\pi x}{10}) \\ u(0) &= 0 \\ u(5) &= 1\end{aligned} \tag{2}$$

This problem has the exact solution $u = \sin(\pi x/10)$.

- Create a file `sine_bvp.py` by copying the program `rope_bvp.py` from the previous lab.
- Simply the signature so that it only includes `nn` with default value 11;
- Inside the function, set `uleft=0.0, uright=1.0`.
- Set the interval to [0,5];
- The matrix entries are simpler this time. Row i is: $A_{i,i-1} = \frac{1}{dx^2}, A_{i,i} = -2 * \frac{1}{dx^2}, A_{i,i+1} = \frac{1}{dx^2}$
- Set b_i from the formula for the right hand side.
- Include a formula for the exact solution:

```
def u_exact ( x ):
  value = ?
  return value
```

- Call `xn, un = sine_bvp(nn)` with `nn = 11`;
- Set `xe` to 101 evenly spaced points, and compute `ue = u_exact ( xe );`
- Plot `(xe,ue)` as a blue line, and `(xn,un)` as red dots together on one plot. If your code is working, these should be very close.

16 Scaling the L^1 and L^2 norms

When we have done convergence studies, we have used the L^∞ norm. Why not the L^1 or L^2 norms instead? We can, except that we first have to factor out an implicit scale factor that depends on the size of the vector. To see what is happening, and how to fix it, suppose we are approximating a function that is equal to 10, and our approximation is 9 everywhere. What is the norm of our error?

The L^infty norm is 1, obviously, no matter how many samples we take. And this is a useful measure of the error.

For the L^1 norm, every entry in the error vector will be 1. So if we take 10 samples, the error norm is 10, if we double to 20 samples, the error is 20, and so on. Just because we looked twice as closely, our error seems to have doubled. But that doesn't seem like a useful report.

Similarly, if we use the L^2 norm, a vector of 10 samples will return an error of $\sqrt{10}$, and when we take 20 samples, our norm will be $\sqrt{20}$, and in general, the error will seem to grow with the factor $\sqrt{n}$.

So if we want to compare errors using different values of n, then L^1 norms need to be divided by the corresponding value of n, and L^2 errors by $\sqrt{n}$. Only then will we be able to see the same kind of convergence behavior that we have already seen when using the L^∞ norm. In the next exercise, we solve a BVP again and again, repeatedly doubling the number of points, and looking at the error behavior using different norms.

17 Exercise 6

1. Create the file `exercise6.py` for this experiment.
2. Set nn as the array [5, 11, 21, 41, 81, 161, 321, 641];
3. For $0 \leq k < 8$
   - Call `xn, un = sine_bvp ( nn[k] )`
   - define `ue = u_exact ( nn[k] )`
   - define `e = ue - un`
   - compute the L^1, L^1/n, L^2, $L^2/\sqrt{n}$ and L^∞ norms of `e`;
4. for $0 \leq k < 7$
   - compute the error convergence rates in each norm.

As a help, you might arrange this calculation as follows:

```
for k in range ( 0, 8 ):
  xn, un = sine_bvp ( nn[k] )
  ue = u_exact ( xn )
  el1[k] = np.linalg.norm ( ue - un, 1 )
  el1n[k] = ?
  el2[k]  = ?
  el2n[k] = ?
  eli[k]  = ?

for k in range ( 0, 7 ):
  rl1 = el1[k] / el1[k+1]
  ...similar commands for rl1n, rl2, rl2n, rli
  print ( k, rl1, rr1, rl2, rr2, rli )
```

The results should convince you that, when computing convergence rates with the L^1 or L^2 norms, you must scale your errors appropriately or you will not see the true rates of convergence.

18 Matrix condition numbers

Given a square matrix A, the L^2 condition number $k_2(A)$ is defined as:

$$k_2(A) = \|A\|_2 \|A^{-1}\|_2 \tag{3}$$

if the inverse of A exists. If the inverse does not exist, then we say that the condition number is infinite. Similar definitions apply for $k_1(A)$ and $k_\infty(A)$ and $k_F(A)$ for the Frobenius norm.

Python can compute the L^p condition number using the command `np.linalg.cond(A,p)`, for `p` = 1, 2, inf, or 'fro'. or just `np.linalg.cond(A )` to use the default L^2 norm. The condition number of a matrix gives us some idea of how "sensitive" the linear system is; it suggests whether a numerical solution is going to be accurate or badly off.

We suppose that we are really interested in solving the linear system

$$Ax = b$$

but that the right hand side we give to the computer has a small error or "perturbation" in it. We might denote this perturbed right hand side as $b + \Delta b$. We can then assume that our solution will be "slightly" perturbed, so that we are justified in writing the system as

$$A(x + \Delta x) = b + \Delta b$$

The question is, if Δb is really small, can we expect that Δx is small? Can we actually guarantee such a limit?

If we are content to look at the *relative errors*, and if the norm used to define $k(A)$ is compatible with the vector norm used, it is fairly easy to show that:

$$\frac{\|\Delta x\|}{\|x\|} \leq k(A) \frac{\|\Delta b\|}{\|b\|}$$

You can see that we would like the condition number to be as small as possible. *(It turns out that the condition number is bounded below. What is the smallest possible value of the condition number?)*. In particular, in typical real arithmetic we have about 16 digits of accuracy, if a matrix has a condition number of 10^{16}, then an error in the last significant digit of any element of the right hand side has the potential to make all of the digits of the solution wrong!

To see how the condition number can warn you about loss of accuracy, our next exercise with try solving the problem $Ax = b$, for `t=np.ones(n)`, with `A` being the Frank matrix. As the size of the matrix grows, so does the condition number. Since we know the exact solution to our problem, we can observe that our computed solution is gradually getting worse and worse.

19 Exercise 7

- Create a file `exercise7.py` for this experiment, with signature

```
def exercise7 ( n, matrix ):
  Input:
    n: the size of the matrix
    matrix: the name of the function that defines the matrix
  ...
  return
```

- Now compute the following:
- `t`, the true solution, a vector of ones, of length `n`;
- `A = matrix ( n )`,

- `c`, the condition number of `A`;
- `b`, the product of the matrix `A` and vector `t`;
- `x`, the numerical solution of `A*x=b`;
- `enorm`, the L^2 norm of `t-x`.
- print `n, c, enorm`

You want to call `exercise7()` with six different matrix sizes `n`, and four different functions `matrix`. Rather than typing 24 separate commands, simplify your life by using the flexibility of python for loops:

```
for matrix in [ helmert_matrix, dif2_matrix, moler3_matrix, frank_matrix ]:
  for n in [ 5, 10, 15, 20, 25, 30 ]:
    exercise7 ( n, matrix )
```

Make several comments about your results:

1. What is the behavior of the condition number `c` for the different matrices?
2. How does the size of `enorm` relate to the size of `c`?
3. Roughly at what value of `c` does the numerical solution seem highly inaccurate?

20 The inverse matrix

In your first class on linear algebra, when discussing the solution of a (square) linear system $Ax = b$, you should have been happy to hear that all your problems would be solved by using the inverse matrix, A^{-1}, which has the properties that $A^{-1}A = AA^{-1} = I$, where I is the identity matrix. That means that we can immediately write an expression for the solution of the linear system:

$$\begin{aligned} Ax &= b \\ A^{-1}Ax &= A^{-1}b \\ x &= A^{-1}b \end{aligned}$$

However, in numerical linear algebra, we **never** do it this way! Instead, if we only have one system to solve, we carry out Gauss elimination directly. If we have many systems to solve, we perform what is called a "PLU" factorization, which gives us something that is equivalent in use to A^{-1}.

So why are we avoiding the computation of A^{-1}?

Except for special matrices, there is usually no formula that will let us write down the inverse matrix immediately. You may have seen one approach, in which you make an $n \times 2n$ matrix $[A|I]$, and then apply full Gauss elimination to the entire system, which will produce the result $[I|A^{-1}]$. However, this approach is expensive, and very prone to numerical roundoff error.

In the next exercise, we will look at the alternatives of solving a linear system by computing the inverse or relying on standard numerical software. We will be interested in the size of the error in the resulting solution.

21 Exercise 8

- Create a file `exercise8.py`, starting from a copy of `exercise7.py`, with signature:

```
def exercise8 ( n, matrix ):
  Input:
    n: the size of the matrix
    matrix: the name of the function that defines the matrix
  ...
  return
```

- Now compute the following:
- `t`, the true solution, a random vector, of length `n`; use `np.random.rand()`;
- `A = matrix ( n )`;
- `c`, the condition number of `A`;
- `Ainv` = inverse of A; use `np.linalg.inv()`;
- `b`, the product of the matrix `A` and vector `t`;
- `x1`, the numerical solution of `A*x1=b`;
- `enorm1`, the L^2 norm of `t-x1`.
- `x2`, the product of `Ainv` and `b`;
- `enorm2`, the L^2 norm of `t-x2`.
- print `n, c, enorm1, enorm2`;

We are comparing the accuracy of two solution methods for a linear system. So now call `exercise8()` with six different matrix sizes `n`, and two different functions `matrix`. Here, we are invoking a new matrix routine called `random_matrix()`.

```
for matrix in [ dif2_matrix, random_matrix ]:
  for n in [ 10, 50, 100, 500, 1000, 2000 ]:
    exercise7 ( n, matrix )
```

It is sometimes estimated that the number of digits of the condition number (to the left of the decimal point) roughly indicates the maximum number of digits of accuracy you might lose in solving a linear system, compared to the maximum of 16. Thus, if $c \approx 10^5$, you might expect your accuracy to drop from 16 digits to 10 or 11. If your true solution is approximately 1, then you might expect your error to be of the order of 10^{-10} or 10^{-11} instead of 10^{-16}.

Can you pick a few of your results for `enorm1` and see whether they agree with this estimate?

As the size of your matrix gets larger, what happens to the size of `enorm2` relative to the size of `enorm1`? Does it seem worth the effort to solve the system using the inverse matrix?

22 Exercise 9

The condition number is based on the product of two matrix norms, and these matrix norms are defined in a very simple way. We can imagine computing $||A||_2$ by taking every possible vector x, computing the ratio $\frac{||A*x||}{||x||}$ and recording the maximum value that occurs. Since we can't actually consider every possible vector, let's do an estimate by looking at lots of them.

Similarly, to estimate $||A^{-1}||_2$, we can repeatedly take a vector at random, and compute the ratio $\frac{||A^{-1}*x||}{||x||}$, and record the maximum value that occurs.

Once we have estimated $||A||_2$ and $||A^{-1}||_2$, their product is our estimate for the condition number of A.

- Create a file `exercise9.py` for this experiment with signature

```
def exercise9 ( n, matrix, tries ):
  Input:
    n: the size of the matrix
    matrix: the name of the function that defines the matrix
    tries: the number of random starting vectors
  ...
  return
```

- Initialize `A = matrix ( n )`, and get `Ainv` from `np.linalg.inv ( )`;
- Initialize `Anorm` and `Ainvnorm` to 0;
- Repeat the following for `k` from 0 to `tries` times:
  - Set `x=np.random.normal(size=n)`;

  - Compute `xnorm = ||x||`;
  - Set `ax` to the product of `A*x`;
  - Compute `axnorm = ||ax||`;
  - Set `Anorm` to the maximum of `Anorm` and `axnorm/xnorm`;
  - Set `ax` by solving `A*ax=x` using `np.linalg.solve()`;
  - Compute `axnorm = ||ax||`;
  - Set `Ainvnorm` to the maximum of `Ainvnorm` and `axnorm/xnorm`;
- Estimate Acond = Anorm*Ainvnorm;
- Print your estimates and exact values of the L2 norms of `A` and `Ainv`, and the condition number of `A`.

You want to call `exercise9()` with six different matrix sizes `n`, but only with the function `dif2_matrix()`. Start with 10 tries. Most of your approximations to norm and condition number will be bad. Try 10 times as many tries. Keep multiplying by 10 until you get to 100,000 tries. Some approximations will get better, but some might not show much improvement.

We are trying to use the definitions of norm and condition number to make an approximation. What can you say about the results as the matrix size `n` increases? Does increasing the value of `tries` result in a good approximation?

23 Exercise 10

In the previous exercise, we tried many random vectors `x` to estimate the norms of A and A^{-1}; for small dimension `n` we came pretty close; for larger dimensions, we had to increase the name of test vectors, and even then our approximations were not great.

Instead of trying lots of random vectors, here's another approach. Start with a random x, and then just keep multiplying it by A over and over. Actually, after we do each multiplication, we'll compute the norm of the vector, and divide the vector by that value, so that we always start with a vector of norm 1. The only thing that changes, then, is the **direction** of the vector. There's a reason to think this might be a good idea, at least some times. So let's try it.

- Create a file `exercise10.py` for this experiment, starting from a copy of `exercise9.py` with signature

```
def exercise10 ( n, matrix, tries ):
  Input:
    n: the size of the matrix
    matrix: the name of the function that defines the matrix
    tries: the number of multiplications
  ...
  return
```

- Initialize `A = matrix ( n )`, and get `Ainv` from `np.linalg.inv ( )`;
- Set `x1` and `x2` each by calling `np.random.normal ( size = n )`;
- Repeat the following for `tries` times:
  - Set `x1` to the product of `A*x1`;
  - Compute `x1norm = ||x1||`;
  - Set `x1 = x1 / x1norm`;
  - Set `x2` by solving `A*x2=x2` using `np.linalg.solve()`;
  - Compute `x2norm = ||x2||`;
  - Set `x2 = x2 / x2norm`;
- Estimate `Anorm = x1norm, Ainvnorm = x2norm, Acond = Anorm*Ainvnorm`;
- Print your estimates and the exact values of the L2 norms of `A` and `Ainv`, and the condition number of `A`.

You want to call `exercise10()` with six different matrix sizes `n`, and four different functions `matrix`. Start with 10 tries to see what happens. See if increasing the value of `tries` improves the results - don't go crazy, don't go beyond 100 tries!

```
tries = 10
for matrix in [ helmert_matrix, dif2_matrix, moler3_matrix, frank_matrix ]:
  for n in [ 5, 10, 15, 20, 25, 30 ]:
    exercise10 ( n, matrix, tries )
```

Just as in the previous exercise, we are trying to estimate matrix norms and condition numbers. But instead of using thousands of random vectors, we just beat one vector to death. You probably got some very good results - at least for some cases. What we are doing is called the *power method*, which we will talk about more later. Our simple version of that method is guaranteed to work well if the matrix is symmetric, but may not produce good results otherwise.

Based on that fact, can you explain why at least one of your approximation efforts never did very well?