Linear Algebra

Peter C. Chapin

February 6, 2010

Contents

Legal

1 Introduction

This document is a quick primer on linear algebra. Its intended audience is senior computer engineering technology students at Vermont Technical College. This document assumes the reader has a reasonable mathematical background, largely calculus based, but no prior exposure to the topics of linear algebra at all.

At the end of each major section below you will find some exercies that will encourage you to practice with the concepts presented in that section. In addition you will find some notes on how to use Octave to experiment with the topics of the section. Octave is an open source program that excels at doing matrix computations of various kinds (as well as many other mathematical things). It is similar to the commerical program MATLAB and even uses a largely compatible syntax. You may also be able to do a number of interesting matrix computations on your calculator. However, Octave will probably have greater capacity, higher performance, and more functionality than your calculator. Regardless of what your calculator can or can't do, I recommend that you spend some time getting familiar with Octave.

2 The Concept of a Matrix

A *matrix* is, essentially, a rectangular array of numbers. It is the basic "data type" used in linear algebra. The individual numbers in a matrix, usually called the *elements*, can be drawn from any of the usual sets. In many applications the matrix elements are real numbers but in some cases they might be complex numbers, integers, or rational numbers. We speak of real matrices, complex matrices, and so forth according to the type of the matrix elements.

The dimensions of a matrix are usually given in the form $m \times n$ where m is the number of rows and n is the number of columns. The row dimension is always given first. If $n = m$ then the matrix is said, for obvious reasons, to be square. Here is an example of how a 2×3 matrix named A might be displayed.

$$A = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{bmatrix}$$

Each element is given two subscripts to indicate its position in the matrix. The first subscript is the row number, ranging from $1 \ldots m$, and the second subscript is the column number, ranging from $1 \ldots n$. Typically when one wants to refer to a generic element of matrix one would use variables such as i, j, or k to be placeholders for the indices. For example, a typical element of matrix A might be written as a_{ij}. As a computer programmer you should be able to relate this notation to the index variables used in typical loops. For example, in C

```
double A[2][3];

for (int i = 0; i < 2; i++) {
  for (int j = 0; j < 3; j++) {
    A[i][j] = ... ;
  }
}
```

Notice that in the mathematical notation it is traditional to regard the matrix subscripts as starting at one and not zero. Also in the mathematical notation it is traditional to use uppercase letters to represent the entire matrix and lowercase letters to represent individual matrix elements.

Matrices with only one row, that is $1 \times n$ matrices, are often called *row vectors*. In many cases the row subscript is dropped, since it is always 1, when talking about the elements of a row vector. Similarly matrices with only one column are often called *column vectors*. In that case the column subscript is often dropped when talking about the elements of a column vector. Row vectors and column vectors are essentially ordinary arrays in a programming sense. In the mathematics of linear algebra they are basically treated like any other matrix although their unusual dimensions give them somewhat special properties.

You might have noticed that in the notation above I did not put any punctuation between the two subscripts in, for example, a_{12}. This might cause you to wonder what happens if one or both of the subscripts exceeds 9. In general this is not a significant concern. In the mathematical development one rarely uses specific numbers other than 1 or 2 anyway. For example, I might show a general $m \times n$ matrix as follows

$$A = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

In the discussion about this matrix I might talk about element a_{ij} and how, for example, it relates to a_{ii} or a_{mj} or even $a_{i+1,j-1}$. However, it would be extremely rare for me to talk about a specific element such as a_{3982} so the ambiguity about what 3982 means in this case would not be an issue.

Keep in mind also that many of the applications of linear algebra involve the use of very large matrices. While the examples in this document and most text books are small so that they can be worked by hand, in the real world matrices with dimensions of many thousands of rows and columns are commonplace. The theory, of course, has no special problems with such large matrices but doing computations efficiently on them can be a serious concern. Much has been written about that subject but it is outside the scope of this document.

Exercises

1. *I need something here.*

Octave

I will assume here that you will be using the open source program Octave for your computer experiments. However, if you have access to MATLAB you should find that all of the commands I talk about here will also work for you. The two programs have a substantial degree of compatibility, particularly in the area of basic commands.

When you start Octave you will get a prompt that might look something like

```
octave:1>
```

The number (shown as "1" above) is the command number. It increments after each command although in this document I will typically show it always as "1." You can use octave as a simple calculator, just type in expressions for it to evaluate. For example

```
octave:1> 2 + 10 / 5
ans = 4
octave:2>
```

Octave stores the result of the evaluation into a special variable named `ans` that it displays on the next line. After displaying the answer it then prints a new prompt. You can store the result of an expression into a named variable using the obvious syntax.

```
octave:1> x = 2 + 10 / 5
x = 4
```

The beauty of Octave is that it allows you to manupulate matrices in a natural way. To enter a matrix use square brackets.

```
octave:1> A = [ 1 2 3; 4 5 6; 7 8 9 ]
A =

  1   2   3
  4   5   6
  7   8   9
```

Octave allows a fairly flexible input syntax. You can separate matrix elements with white space alone or with commas at your option. The semicolons are used to delimit one row from another. However, you could have also entered the matrix on multiple lines using the RETURN key at the end of each row in a natural way.

Once a matrix has been entered you can access an individual element using parenthesis. For example `A(1,3)` prints out the value of a_{13}. You can also modify a particular matrix element.

```
octave:2> A(1,3) = 10
A =

  1   2  10
  4   5   6
  7   8   9
```

Notice that when a matrix element is modified, Octave redisplays the entire matrix.

There are some quirks with Octave to keep in mind as you use it. First, as is mathematically traditional, element indices are numbered starting at 1, not 0 as a C programmer might expect. Also Octave uses parenthesis instead of square brackets when accessing an individual element. These things are consistent with Fortran's syntax and are considered natural among numerical specialists who use Fortran. Also be aware that Octave uses a floating point type internally to hold all scalars. This is typically what you want but it means that its integer calculations are somewhat of an illusion. Be careful.

You should take Octave for a test drive. Try entering in a few matrices and accessing their elements. Although I didn't discuss it here, experiment with Octave's slice notation. For example, using the matrix A entered above, try executing the command `A(2:3, 2)`. Can you explain what happens? Verify your explaination by trying some other possibilities. Also try `A(:, 2)` as well.

3 Matrix Operations

Various operations have been defined for matrices so that they may be manipulated as a single unit. In this section I will review some of those basic operations. Please read this section carefully. Understanding this material is essential for understanding the later material.

Two matrices can be added if they have the same dimensions. If the dimensions are different the addition of the matrices is undefined. To add two compatible

matrices one simply adds corresponding elements. Thus addition is very natural. For example

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} + \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} a_{11} + b_{11} & a_{12} + b_{12} \\ a_{21} + b_{21} & a_{22} + b_{22} \end{bmatrix}$$

To multiply a matrix by a scalar, simply multiple each element in the matrix by that scalar. This is also very natural. For example

$$a \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} ab_{11} & ab_{12} \\ ab_{21} & ab_{22} \end{bmatrix}$$

Matrix subtraction follows from the above two operations. If A and B are matrices then $A - B$ can be regarded as $A + (-1)B$. Thus matrix subtraction is also done on an element by element basis in a natural way.

Because matrix addition is defined as element by element addition it should be fairly clear that matrix addition obeys the same laws of commutivity and associativity as ordinary addition. That is, if A, B, and C are matrices then $A + B = B + A$ and $(A + B) + C = A + (B + C)$. This assumes that all the matrices involved have the same dimensions so that the addition operations are properly defined.

Matrix multiplication, on the other hand, is defined in a way that might seem rather counter-intuitive at first. However, as you will see later, it actually provides a very useful operation. Let the matrices to be multipled be called A and B. Let $C = AB$ be the product matrix. Let A have the dimensions $m \times n$ and B have the dimensions $n \times p$. For the multiplication to be defined the two inner dimensions must agree. The dimension of C is then given by the two outer dimensions. In this case that would be $m \times p$. For example you could multiple a 3×2 matrix with a 2×5 matrix. The result would be a 3×5 matrix.

To compute an element in the result matrix, say c_{ij}, you would run down row i of A and column j of B. Because of the rules described above, there would be n columns in A and n rows in B so the number of items in each run would be the same. You calculate c_{ij} as follows

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

The computation is, essentially the vector dot product of the i^{th} row vector of matrix A and the j^{th} column vector of matrix B. Of course you would have to repeat this calculation, using different rows and columns in A and B respectively, for each element in the result matrix.

As an example consider the following product of a 3×2 matrix with a 2×3 matrix.

$$\begin{bmatrix} 1 & 2 \\ 4 & -1 \\ -3 & 3 \end{bmatrix} \times \begin{bmatrix} -2 & 3 & 1 \\ 5 & 2 & 5 \end{bmatrix} = \begin{bmatrix} 8 & 7 & 11 \\ -13 & 10 & -1 \\ 21 & -3 & 12 \end{bmatrix}$$

For example $c_{11} = a_{11}b_{11} + a_{12}b_{21} = (1)(-2) + (2)(5) = 8$. The other elements of the result matrix are calculated similarly. You should verify them to be sure you understand how matrix multiplication works. Notice also that in this case the result matrix is larger than either of the matrices being multiplied together. This is a consequence of dimensioning rules. It won't always be like that, however. For example a 2×6 matrix being multiplied with a 6×2 matrix will result in a smaller 2×2 matrix. In the special case where all the matrices involved are square the dimensions will stay the same during the multiplication.

Matrix multiplication is not commutative. For one thing reversing the order of the matrices might cause the multiplication to be undefined due to incompatible dimensions. Even if the multiplication is defined the result matrix might not be the same size. For example, reversing the order of the matrices in the last example will produce a 2×2 result, not a 3×3 result. Even if the result matrices are the same size (both A and B are square), the results will not, in general, be the same. It is easy to find two square matrices that don't commute when multiplied. Most pairs don't.

Matrix multiplication does, however, associate. That is if A, B, and C are matrices $(AB)C = A(BC)$. As a consequence of this the parenthesis are not necessary when writing the product of several matrices. It also means that something like A^3 is well defined. In particular, $A^3 = AAA = (AA)A = A(AA)$. Notice that only square matrices can be raised to powers (do you see why?).

At this point you might be wondering about the definition of matrix division. It turns out that the problem of dividing matrices is fraught with subtle complications. In fact we do not define it at all in the usual sense. Instead we talk about matrix inverses. However, that is a topic that deserves its own section.

Exercises

1. Demonstrate that matrix multiplication, even of square matrices, is not in general commutative.

2. Prove that matrix multiplication is associative.

3. Does matrix multiplication distribute over matrix addition? That is, if A, B, and C are matrices does $A(B + C) = AB + AC$? If so, then prove it. If not, then find a counter example using Octave (see below). Repeat the question for $(B + C)A = BA + CA$.

Octave

Once matrices are defined in Octave you can add, subtract, and multiply them with the obvious expressions. Octave will print the results of each expression as you enter it (you can supress such printing by ending your command with a semicolon). Use Octave to demonstrate the commutativity and associativity (or lack thereof) of matrix addition and matrix multiplication. Use Octave to support your answer to the distribution question above. Either find suitable counter examples or demonstrate that distribution does work.

Keep in mind that numerical experiments of the sort you can do in Octave can prove that a certain property does not hold in all cases by finding a single counter example. You do not need to try every matrix in existence. However, Octave can not prove that a certain property does hold in all cases since it can't test every possible case. One must rely on the standard techniques of mathematical proof to verify that a property is true in all cases.

More specifically, you can use Octave to prove that matrix multiplication does not commute by finding two matrices that don't commute. However, you can't prove that matrix multiplication is associative without computing $A(BC)$ and $(AB)C$ for every possible A, B, and C. That would require an infinite amount of computation since there are infinitely many matrices. However, you can *demonstrate* the association of matrix multiplication in Octave by computing $A(BC)$ and $(AB)C$ for some particular A, B, and C and then showing that the results are the same. Such a demonstration does not constitute a proof of the general case but it is interesting anyway.

4 Linear Transformations

Exercises

1. *I need something here.*

Octave

5 Matrix Inverses

In order to talk about the inverse of a matrix I should first talk about the concept of identities. Consider ordinary multiplication. There exists a number, namely 1, that has the property $x \times 1 = 1 \times x = x$. Thus 1 is called the *multiplicative identity* because when you multiply any number by 1 the result is the same. It

turns out that there is a *identity matrix* as well. Actually there are infinitely many identity matrices of different sizes. The general form is

$$I_n = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 \end{bmatrix}$$

That is, I_n is an $n \times n$ square matrix that has ones down the diagonal and zeros everywhere else. It is relatively easy to see that $IA = A$ for any matrix A (just try multiple out an example and you will quickly see how it works). Note that A need not be square. As long as you use the proper sized identity matrix so that the multiplication is defined the result of IA will be A again.

The identity matrix is also special because, unlike the general case, multiplication by the identity martix commutes. That is $IA = AI = A$. This assumes that A is square. Otherwise one must use different sized identities depending on if the identity is being pre-multiplied or post-multiplied.

Another property of real numbers is that every number except zero has a *multiplicative inverse*. For example x's inverse, symbolized as x^{-1} is such that $xx^{-1} = x^{-1}x = 1$. Similarly many matrices have inverses as well. For example A's inverse, symobolized as A^{-1} is such that $AA^{-1} = A^{-1}A = I$. Here I restrict myself to square matrices. In that case notice how multiplication by the inverse commutes; again contrary to the general case for matrix multiplication.

With real numbers we can define division in terms of multiplication by an inverse. Specifically we can define a/b as ab^{-1} or as $b^{-1}a$. Here I'm taking advantage of the fact that multiplication of real numbers commutes in every case. Similarly in situations where one is tempted to divide by a matrix we normally talk instead about multiplying by the inverse of the matrix instead. Because matrix multiplication is not commutative we have the added complication, in the algebra of matrices, of having to carefully track pre-multiplications as opposed to post-multiplications. An example of this will be shown in the next section.

Computing the inverse of a matrix is a non-trivial problem. I will discuss this more in a later version of this document.

Exercises

1. Consider the following $n \times n$ matrix.

$$A = \begin{bmatrix} a_{11} & 0 & \cdots & 0 \\ 0 & a_{22} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & a_{nn} \end{bmatrix}$$

This matrix has non-zero elements only on its main diagonal. What is its inverse?

Octave

You can use the `eye()` function in Octave to generate an identity matrix of any size. For example `eye(3)` returns a 3×3 identity. You can also use the `inv()` function to compute the inverse of a matrix. For example

```
octave:1> A = [ 1 2 4; -4 2 1; 9 -2 1 ];
octave:2> B = inv(A)
```

Verify that the result of `inv()` is a proper inverse by computing $A*B$ and $B*A$ and showing that both result in the identity matrix. You may notice that the results shows slight round off errors in the computations (mostly identified by the existence of negative zeros in the result). This is normal.

What does Octave do when asked to find the inverse of the singular matrix in the example above?

6 Matrix Norms

Before I can discuss the norm of a matrix, I should first describe the norm of a vector. A vector's *norm* is a measure of the vector's length. For example, if the components of a vector X are x_1, x_2, and x_3 the usual formula for the vector's length is $L(X) = \sqrt{x_1^2 + x_2^2 + x_3^2}$. This formula has the following "length-like" properties.

1. $L(X) \geq 0$, with $L(X) = 0$ if and only if $X = 0$.
2. $L(X + Y) \leq L(X) + L(Y)$.
3. $L(\alpha X) = |\alpha| \, L(X)$.

The second property above is called the triangle inequality. It says that the length of the sum of two vectors is less than or equal to the sum of the two lengths. A simple diagram will make it obvious that this is true in the case of the traditional concept of length.

The vector function $L(X)$ above satisfies the length-like properties but it is not the only function that does so. Any function that satisfies those properties (for all vectors) can be considered a "length". Such functions are norms. In

fact, $L(X)$ above is actually one function in an entire class of functions called *p-norms*. The general form of a p-norm is

$$\|X\|_p = (|x_1|^p + |x_2|^p + \cdots + |x_n|^p)^{\frac{1}{p}},\ p \geq 1$$

We use the symbol $\|X\|_p$ to represent the p-norm of vector X. You can see that the traditional formula for the length of a vector is the 2-norm of the vector. In addition to the 2-norm we will also be interested in the 1-norm and the ∞-norm. The ∞-norm is the result of a limiting process and is found from

$$\|X\|_\infty = \max_{1 \leq i \leq n} |x_i|$$

That is the ∞-norm is simply the component of the vector with the largest absolute value.

We can now extend these concepts to matrices. One way to do that would be to treat a $m \times n$ matrix as a vector with mn components. One could then simply use a suitable vector norm as a matrix norm. It is easy to see that a matrix norm defined in this way will continue to have all the required properties provided that the underlying vector norm has those properties.

However, since the product of two $n \times n$ matrices is again an $n \times n$ matrix, it turns out to be useful to add an additional constraint on the behavior of matrix norms. In particular, we will want $\|AB\| \leq \|A\| \cdot \|B\|$. With this condition in place the simple extension of vector norms to matrix norms does not work.

However, one can define useful matrix norms in terms of vector norms anyway. In particular, let the norm of a matrix A be

$$\|A\| = \max_{\|u\|=1} \|Au\|$$

To understand what this means remember that a matrix, when multiplied by a vector, performs a linear transformation of that vector into another vector. In the above expression the vector u has a norm of 1 and is thus a *unit vector*. There are, in general, infinitely many unit vectors (depending on the norm used). For example, if one uses the 2-norm the unit vectors describe a sphere of radius 1 centered on the origin. The norm of a matrix is thus the norm of the "largest" vector produced by applying the matrix as a linear transformation to every possible unit vector. For example, if one used a 2-norm and applied the matrix to every point on the surface of a unit sphere, the norm of the matrix would be the distance from the origin to the most distant point on the resulting surface.

It is important to note that any vector norm can be used to define a matrix norm in this way. Then when one speaks, for example, of the p-norm of a matrix one is talking about the matrix norm that results from using the corresponding

p-norm of a vector. This method of defining a matrix norm may seem odd but it has a number of useful properties. For an $m \times n$ matrix

$$\begin{aligned} \|A\|_1 &= \max_{1 \le j \le n} \sum_{i=1}^{m} |a_{ij}| \\ \|A\|_\infty &= \max_{1 \le j \le m} \sum_{j=1}^{n} |a_{ij}| \end{aligned}$$

In other words the 1-norm is the maximum of the sums of the columns, with the understanding that it is the sums of the absolute values of the elements that are done. Similarly the ∞-norm is the maximum of the sums of the rows, with the same understanding about taking the absolute values of the elements. For example, the matrix below shows the column sums below each column and the row sums beside each row.

$$\begin{array}{rrrcr} \multicolumn{3}{c}{\left[\begin{array}{rrr} 1.23 & -3.42 & 0.89 \\ -4.48 & 4.56 & 6.22 \\ 5.19 & 3.95 & -2.88 \end{array}\right]} & \begin{array}{c} \rightarrow \\ \rightarrow \\ \rightarrow \end{array} & \begin{array}{r} 5.54 \\ 15.26 \\ 12.02 \end{array} \\ \downarrow & \downarrow & \downarrow & & \\ 10.90 & 11.93 & 9.99 & & \end{array}$$

Thus the 1-norm of the matrix above is 11.93 and the ∞-norm of the matrix is 15.26. Notice that if a single element in the matrix was much larger than any of the others, it would dominate its column sum and its row sum. That element would become, in effect, both the 1-norm and the ∞-norm.

It is important to realize that both the 1-norm and the ∞-norm of a matrix can be calculated in $\Theta(n^2)$ time. As you will see in Section 7, this is more efficient than what is required to invert a general matrix. This observation is important during the solution of simultaneous equations.

Exercises

1. Calculate the 1-norm, 2-norm, 3-norm, and ∞-norm of the following vectors

   (a) $(1.27, -3.12, 4.84)$

   (b) $(3.44, -2.19, 15.32)$

   (c) $(0.00, 0.00, 0.00)$

2. Prove that for any vector X, $\|X\|_\infty \le \|X\|_p$ for all finite p. Prove that if $q > p$, then $\|X\|_q \le \|X\|_p$.

3. Calculate the 1-norm and ∞-norm of the following matrix.

$$\begin{bmatrix} 6.28 & -12.31 & 4.22 & 1.33 \\ 4.89 & -2.23 & 6.69 & -3.44 \\ 8.32 & 2.41 & -10.34 & 9.83 \\ 0.48 & -0.48 & -18.40 & 4.44 \end{bmatrix}$$

4. How do the p-norms of the transpose of a matrix relate to the p-norms of the original matrix?

Octave

Octave has a built in `norm` function that can be used to compute the norm of a vector or of a matrix. Read about it in Octave's online help. Use that function to check your answers in Exercise 1 and Exercise 3 above. Use Octave to compute the norms of several matrices and their transposes and see if you can verify your findings in Exercise 4.

7 Simultaneous Equations

One of the most important applications of matrices and linear algebra is in the study of systems of simultaneous linear equations.

A system of two linear equations and two unknowns can always be written in the following general way.

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 &= b_1 \\ a_{21}x_1 + a_{22}x_2 &= b_2 \end{aligned}$$

The unknowns are x_1 and x_2. In many elementary texts the unknowns are often referred to as x and y but that notation is less extensible to the more general case of many unknowns. The equations above are called *linear equations* because each equation taken alone defines a straight line on an x_1 vs x_2 plane. The intersection of those two lines defines a single point, (x_1, x_2), that satisfies both equations simultaneously.

One is usually interested in solving the system of equations for the two unknowns given values for the coefficents and for b_1 and b_2. However, before worrying about how to best solve such systems, it is useful to make the following observation. Let

$$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$$

Call x the *vector of unknowns*. Notice that it is a 2×1 column vector. Let

$$b = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

Call b the *driving vector*. It is also a 2×1 column vector consisting of the constant terms in the system of equations. Finally let

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix}$$

Call A the *matrix of coefficients*. It is a 2×2 square matrix.

Now the system of equations can be succintly expressed by the matrix equation

$$Ax = b$$

The expression Ax is the multiplication of a 2×2 matrix with a 2×1 column vector. By the rules of matrix multiplication this is properly defined. The result will be another 2×1 column vector—exactly the dimensions of b. The first element in b, in row 1, column 1, can be found by scanning down row 1 of A and column 1 of x and forming the sum $a_{11}x_1 + a_{12}x_2$. Thus

$$b_1 = a_{11}x_1 + a_{12}x_2$$

This is, of course, the first equation in the system of equations. The other equation follows similarly. Now we can see how the odd definition of matrix multiplication can be useful!

A linear system of three equations and three unknowns can be written

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= b_1 \\ a_{21}x_1 + a_{22}x_2 + a_{23}x_3 &= b_2 \\ a_{31}x_1 + a_{32}x_2 + a_{33}x_3 &= b_3 \end{aligned}$$

Each equation in such a system defines a plane in three dimensional space with coordinates x_1, x_2, and x_3. Two planes intersect to define a line and the intersection of that line with the third plane defines a single point. That point is the solution of the system; it is the specific value of (x_1, x_2, x_3) that satisfies all three equations simultaneously. Although this system is larger, it can still be expressed with the matrix equation $Ax = b$ with the understanding that A is a 3×3 matrix in this case.

What of very large matrices? Suppose A was a general $n \times n$ matrix. In that case each row in A would correspond to a single equation with n unknowns. Such an equation defines a $n - 1$ dimensional hyperplane in the n dimensional space with coordinates $x_1, x_2, \ldots, x_n$. The intersection of two $n-1$ dimensional hyperplanes defines a $n - 2$ dimensional hyperplane. The intersection of that

$n-2$ dimensional hyperplane with another $n-1$ dimensional hyperplane results in a $n-3$ dimensional hyperplane. As each equation is used, the dimension of the region of intersection decreases by 1. When all n equations are considered, the region of intersection is a zeroth dimensional region—a point. It is the point $(x_1, x_2, \ldots, x_n)$ that simultaneously satisfies all the equations. Even if n is very large, the system of equations can still be represented by the simple form $Ax = b$.

Solving $Ax = b$ is, in theory very simple. Just "divide" both sides by A. More precisely

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

In particular, we can pre-multiply both sides of the equation by A^{-1} and then use the associativity of matrix multiplication and the special properties of the identity matrix to reduce the left hand side of the equation to just x. The right side of the equation then becomes $A^{-1}b$. Two matrices are equal only if their corresponding elements are all equal. Thus the first element of the x column vector, x_1, must be equal to the first element in the $A^{-1}b$ column vector. This is the solution for the unknown x_1. The other unknowns are given by the other values in the $A^{-1}b$ column vector.

Thus we have this important result: Solving the system of equations is simply a matter of finding the inverse of the matrix of coefficients, if it exists, and then pre-multiplying the driving vector by that inverse. Notice that if the driving vector is changed a new solution can be found with a simple matrix multiplication. The matrix of coefficients does not need to be re-inverted unless one of A's elements changes. This can be significant in some applications. As you will see, the driving vector is usually an expression of signals applied to a system while the matrix of coefficients is usually related to the structure of a system. Computing what a system does with different inputs is usually a matter of solving the equations with a new driving vector. Once the coefficients are inverted this is a simple matter.

You know from Section 5 that not all matrices have inverses. What is the significance of the case when the matrix of coefficients is singular? In that situation the system of equations has no solution. However, the physical interpretation of this case is quite interesting and worth looking at more closely. Consider the following system

$$\begin{bmatrix} 1 & 2 \\ 1 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 3 \\ 4 \end{bmatrix}$$

The two lines defined by these equations are parallel; they have no point of

intersection[1]. Thus there is no point (x_1, x_2) that can satisfy both equations at the same time. It also happens that the matrix of coefficients is singular. Now consider the system

$$\begin{bmatrix} 1 & 2 \\ 2 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$$

Here one equation is simply a scaled version of the other. As a result they are really the same equation. This system does not contain enough information to have a unique solution. In effect it has only one equation. Once again the matrix of coefficients is singular.

Both of these examples have one thing in common. One row of the matrix of coefficients is a scaled version of another row. In the first example the scaling factor was 1. In the second example the scaling factor was 2. In the first example the values in the driving vector did not use the same scaling factor and the result was parallel lines. In the second example the values in the driving vector did use the same scaling factor and the result was two identical lines (which are also parallel, of course).

This observation can be generalized but it is necessary to first define a new concept. Suppose u and v are n element row vectors. A *linear combination* of u and v is any vector, w such that

$$w = a_1 u + a_2 v$$

where a_1 and a_2 are any arbitrary scalars. In other words, I can express w in terms of u and v by scaling each vector and adding the scaled results. Note that a_1 and a_2 could be negative or zero. If w is a linear combination of u and v then we say that w is *linearly dependent* on u and v. If you have a vector that can not be expressed as a linear combination of other vectors we say that it is *linearly independent* of those other vectors.

Consider two vectors in a plane that don't point along the same line. There is no way I can scale one to get the other. Thus the two vectors are linearly independent. However, any other vector in that plane can be expressed as a linear combination of the first two vectors. Thus any other vector in that plane is linearly dependent of the first two. Consider the set of all vectors that can be written as a linear combination of the first two. That set of vectors constitutes a *vector space* and the first two, linearly independent vectors, form the *basis* of that space. They are said to *span* the space.

For any particular vector space there are many different sets of basis vectors possible. Any two linearly independent vectors in a plane can be used as a basis for that plane. It is common in engineering and physics to use two orthagonal[2] vectors of length 1 as a basis for the plane. However, it is perfectly possible, in general, to use two oblique vectors of arbitrary lengths as a basis as well.

[1] This might be easier to see if you rearrange each equation into the form $x_2 = mx_1 + b$

[2] perpendicular

Now suppose that you introduce a third vector that points out of the plane spanned by the first two. There is no way to write this third vector as a linear combination of the others and so it is linearly independent of them. In fact, we can talk about the space spanned by the three vectors—the first two and this new one. That space has three dimensions. Hopefully you can see that the number of basis vectors required to span a space is equal to that space's dimensionality. In fact, this is the where the very concept of dimensionality comes from.

Let's bring this back to the subject of simultaneous equations. Each equation in an $n \times n$ system defines a $n-1$ dimension hyperplane. In particular, if you regard each equation as a row vector, the hyperplane that it describes is perpendicular to that vector and passes through the tip of that vector. If any of the hyperplanes are parallel there can be no solution to the system. Furthermore if the intersection of any combination of those hyperplanes is parallel to any of the remaining hyperplanes there can be no solution. This will occur if any row of the matrix of coefficients can be written as a linear combination of the other rows. In that case, the system contains insufficient information for a unique solution. The matrix of coefficients will be singular.

The following is important.

Theorem 7.1 *Let A be an $n \times n$ matrix. A has an inverse if and only if* $\det(A) \neq 0$. *Furthermore A has an inverse if and only if the rows of A are linearly independent.*

Since A needs to have an inverse in order to solve the matrix equation $Ax = b$, the theorem above is directly relevant to solving such systems. However, in general computing the $\det A$ is expensive so a somewhat different approach that is more practical is usually taken.

In a realistic computation the coefficients in the matrix of coefficients are not normally known to infinite precision. If they represent physical quantities they may only be accurate to two or three significant figures. Even if they coefficients are known to infinite precision, the computer used to calculate a solution has finite precision. Thus you would not expect the solution to be perfectly accurate.

If the matrix of coefficients is close to being singular that means that the hyperplane described by one of its rows is nearly parallel to the intersection of the other hyperplanes. As a result, very slight variations in the coefficients will cause a very large change in the solution. Such a system is said to be *ill-conditioned*. In real calculations, exactly singular matrices don't exist. Singularity is a matter of degree and, due to practical limits of precision, one normally only sees systems with relatively more or less ill-conditioning.

Since many naive formulations of real problems lead naturally to ill-conditioned systems, it is important to be aware of this issue and to check the conditioning

of one's system before expending too many computational resources trying to solve it. Thus we can define a *condition number* of a matrix, κ_∞, as a figure of merit that we can use to judge a system's degree of ill-conditioning.

$$\kappa_\infty(A) = \|A\|_\infty \cdot \|A^{-1}\|_\infty$$

In a perfectly conditioned case, all the hyperplanes described by the rows of the matrix are perpendicular. This occurs for the identity matrix and thus $\kappa_\infty(I_n) = 1$ is the ideal. For matrices where some of the hyperplanes are oblique κ_∞ is greater than one and approaches infinity for the singular case.

Note that you can compute the condition number of a matrix using a norm other than the ∞-norm. However, the general properties of the condition number remain the same. One reason why κ_∞ is nice to use is because the ∞-norm of a matrix is easy to calculate (see Section 6). Unfortunately a proper computation of κ_∞ requires that the matrix be inverted. However, if the system is ill-conditioned, the inverse of A can't be computed accurately. Indeed, the whole point of computing κ_∞ is to know if A^{-1}, or an equivalent result, can be calculated with reasonable accuracy. Much has been written on how to estimate a system's condition number without computing the inverse of the matrix of coefficients and without spending too much time doing it. A full discussion of this matter is outside the scope of this short document.

Exercises

1. *I need something here.*

Octave

8 Gaussian Elimination

Although this document is not intended to cover issues related to numerical computation, there is one numerical method that is so important that it does deserve coverage: Gaussian Elimination. This method can be used to solve a system of linear equations in a reasonably effective manner. Although it requires a running time of $O(n^3)$ it is considerably more efficient than some other techniques (such as Kramer's Rule using determinants). In addition Gaussian Elimination has good numerical properties and works well on dense systems[3].

In principle Gaussian Elimination is very simple to understand. Three basic row operations are defined.

[3] A *dense* system of equations is one in which a large percentage of the coefficients are not zero.

1. Exchange two rows, $R_i \leftrightarrow R_j$
2. Multiple a row by a scalar, $R_i \leftarrow aR_i$
3. Replace a row with the sum of that row and another scaled row, $R_j \leftarrow R_j + aR_i$

In each case it is necessary to also apply the operations to the corresponding elements of the driving vector. It is easy to see that these operations are justified based on concepts from elementary algebra. For example exchanging two rows is equivalent to writing down the equations in a different order; this does not change the solution of the system. Furthermore, multiplying a row by a scalar is equivalent to multiplying an equation by a scalar, and so forth.

The easiest way to explain how Gaussian Elimination works is to show an example. Consider the following 3x3 system of equations.

$$\begin{array}{rcrcrcr} 1.23\,x_1 & - & 4.53\,x_2 & + & 2.83\,x_3 & = & 6.77 \\ 8.33\,x_1 & + & 1.93\,x_2 & + & 3.28\,x_3 & = & -2.33 \\ -3.48\,x_1 & + & 7.12\,x_2 & - & 1.20\,x_3 & = & 6.12 \end{array}$$

We can write this system in matrix form as follows

$$\begin{bmatrix} 1.23 & -4.53 & 2.83 \\ 8.33 & 1.93 & 3.28 \\ -3.48 & 7.12 & -1.20 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 6.77 \\ -2.33 \\ 6.12 \end{bmatrix}$$

The first phase of the computation is the *elimination* phase. It entails using the three row operations listed above to change the matrix of coefficients into a upper triangular matrix. In other words, a matrix in which all the elements below the main diagonal are zero. The computation proceeds by considering each diagonal element one at a time. Starting with a_{11} we start be scanning down the column below and including the diagonal element looking for the element with the largest absolute value. In this case row two holds that element ($a_{21} = 8.33$). We thus do $R_1 \leftrightarrow R_2$ to exchange those rows.

$$\begin{bmatrix} 8.33 & 1.93 & 3.28 \\ 1.23 & -4.53 & 2.83 \\ -3.48 & 7.12 & -1.20 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -2.33 \\ 6.77 \\ 6.12 \end{bmatrix}$$

This step is not strictly necessary but it turns out to improve the accuracy of the computation. Notice that the elements in the driving vector are exchanged as well. This is necessary, of course, because those elements are part of the equations being exchanged.

Next using a_{11} we do row operations to zero out the coefficients below it in the same column. Specifically we do $-(1.23/8.33)R_1+R_2 \rightarrow R_2$ and $(3.48/8.33)R_1+$

$R_3 \rightarrow R_3$. The result is

$$\begin{bmatrix} 8.33 & 1.93 & 3.28 \\ 0.00 & -4.82 & 2.35 \\ 0.00 & 7.93 & 0.17 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -2.33 \\ 7.11 \\ 5.15 \end{bmatrix}$$

Moving on to a_{22} we again search the column beneath and including that element looking for the element with the largest absolute value. In this case that element is $a_{32} = 7.93$. We thus do $R_2 \leftrightarrow R_3$ to exchange those rows.

$$\begin{bmatrix} 8.33 & 1.93 & 3.28 \\ 0.00 & 7.93 & 0.17 \\ 0.00 & -4.82 & 2.35 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -2.33 \\ 5.15 \\ 7.11 \end{bmatrix}$$

Finally we zero out the coefficients below the (new) a_{22} by doing $(4.82/7.93)R_2 + R_3 \rightarrow R_3$.

$$\begin{bmatrix} 8.33 & 1.93 & 3.28 \\ 0.00 & 7.93 & 0.17 \\ 0.00 & 0.00 & 2.45 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} -2.33 \\ 5.15 \\ 10.24 \end{bmatrix}$$

This completes the elimination phase. The next phase is called *back substitution*. We start by observing that the last row represents the equation $2.45\,x_3 = 10.24$. From this equation we can immediately calculate $x_3 = 10.24/2.45 = 4.18$. Notice also that the second to last row can be written as $7.93\,x_2 + 0.17\,x_3 = 5.15$. Of course this equation can be written as $x_2 = (5.15 - 0.17\,x_3)/7.93$. Given that x_3 has just been calculated we can now compute $x_2 = 0.56$. Similarly $x_1 = (-2.33 - 1.93\,x_2 - 3.28\,x_3)/8.33 = -2.06$.

The back substitution phase can store the computed results in the space used for the driving vector. After each value is computed the corresponding driving vector slot is no longer needed and can be used for storing the final result.

If you substitute the values for (x_1, x_2, x_3) computed above back into the original system you will find that they *almost* work. The error you observe is due to the imprecise nature of the computations done. In this example only two or three significant figures are retained in each step. As with any numerical computation, errors can potentially accumulate as the computation proceeds. In misbehaving situations, the resulting accumulated errors can be so great as to swamp out any real results. Because it is such an important algorithm, Gaussian Elimination has been extensively studied with respect to its error propagation behavior. However, it is outside the scope of this document to discuss that issue any further.

It is important to note that the elimination phase of the computation runs in $O(n^3)$ time, where n is the number of equations being solved. This is because each diagonal element must be considered and, for each such element an average of $n/2$ rows must be processed and, for each row $O(n)$ computations are needed.

In contrast, the back substitution phase only requires $O(n^2)$ time. This is because n rows must be processed and, for each row an average of $O(n/2)$ computations are needed. This means for large systems, the dominate factor limiting the performance of the algorithm is the elimination phase.

Exercises

1. *I need something here.*

Octave

9 Eigenvalues and Eigenvectors

Exercises

1. *I need something here.*

Octave