

Physics 305: Ch. 1

Linear Algebra

1.1 Introduction to Linear Algebra

Linear algebra is the study of linear operations on vector spaces. By a linear operation, we mean a function $A(\vec{x})$ with the properties that $A(c\vec{x}) = cA(\vec{x})$ for any number c and $A(\vec{x} + \vec{y}) = A(\vec{x}) + A(\vec{y})$ for two vectors \vec{x} and \vec{y} in the same vector space. Often we consider that the function A is itself yielding vector values, which may be vectors in the same space as \vec{x} or in another space.

More concretely, if we consider functions from a M -dimensional vector space of \vec{x} to a N -dimensional vector space of \vec{b} , then the most general linear function has the form

$$b_1 = a_{1,1}x_1 + a_{1,2}x_2 + \dots + a_{1,M}x_M \quad (1.1)$$

$$b_2 = a_{2,1}x_1 + a_{2,2}x_2 + \dots + a_{2,M}x_M \quad (1.2)$$

$$\dots \quad (1.3)$$

$$b_N = a_{N,1}x_1 + a_{N,2}x_2 + \dots + a_{N,M}x_M \quad (1.4)$$

where x_i and b_j are the coordinates in the vectors and $a_{j,i}$ are constants. You should convince yourself that this set of equations is in fact linear in the sense described above.

We then abbreviate this with matrix notation as $\vec{b} = \mathbf{A} \cdot \vec{x}$, where we arrange the coefficients $a_{i,j}$ into a matrix \mathbf{A} . We use

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1N} \\ a_{21} & a_{22} & \dots & a_{2N} \\ \cdot & \cdot & \dots & \cdot \\ a_{N1} & a_{N2} & \dots & a_{NN} \end{bmatrix}, \quad \vec{x} = \begin{bmatrix} x_1 \\ x_2 \\ \cdot \\ x_N \end{bmatrix}, \quad \vec{b} = \begin{bmatrix} b_1 \\ b_2 \\ \cdot \\ b_N \end{bmatrix}$$

The rule for multiplying the vector \vec{x} by the matrix \mathbf{A} is then just

$$\mathbf{A} \cdot \vec{x} = \sum_{i=1}^N a_{ij}x_j.$$

On the computer, this would just be two loops; an inner loop to do the summation, and an outer loop to fill all of the elements of \vec{b} :

```

for i = 1 to N
  b_i = 0
  for j = 1 to M
    b_i = b_i + a_{ij} x_j
  end
end
end

```

Note that in this section, we have labeled elements $1 \dots N$ as the mathematicians do, but in C you will do everything zero-indexed, i.e., $0 \dots N - 1$.

There are a wide range of interesting physics, mathematics, and data reduction problems that can be formulated as linear algebra problems. In this course, we'll focus on two of the most common situations.

1) Solving a system of linear equations, i.e., finding \vec{x} in $\mathbf{A} \cdot \vec{x} = \vec{b}$ when given \mathbf{A} and \vec{b} . Here the two vector spaces have the same dimensionality (i.e., $M = N$), so \mathbf{A} is called a square matrix. This problem is equivalent to finding the inverse of the matrix \mathbf{A} .

2) Finding the eigenvectors and eigenvalues of a matrix \mathbf{A} , i.e. the special vectors \vec{x} for which $\mathbf{A} \cdot \vec{x} = \lambda \vec{x}$ for some numbers λ . Again, \mathbf{A} is a square matrix. We will be particularly interested in the case where \mathbf{A} is also a symmetric matrix of real numbers; in this case, the eigenvalues are all real numbers.

1.2 Computational Linear Algebra

The topic of solving these and other linear algebra problems is an enormous subject in numerical computation. Linear systems are often quite large, and the general solutions to these problems scale as the cube of the order of the matrix. Note that a matrix of 10,000 by 10,000 elements requires nearly a gigabyte to store (in double precision) and would take many trillions of operations to invert (i.e., hours to days). There has been a lot of work on how to achieve high computational efficiency and/or high computational accuracy. Special classes of matrices often permit savings, sometimes factors of two, sometimes factors of thousands. Another major topic is what to do when your matrix is truly enormous (even bigger than your computer can hold) but with only a small number of non-zero elements; these are called sparse matrices. Iterative methods exist for these that permit approximate solutions in far less than the expected amount of time.

There are significant repositories of codes to do linear algebra problems; you usually don't need to write your own. However, it does pay to become familiar with the subject so that you know what is possible. It can be an enormous speed-up if you can transform your problem into linear algebra! The book Numerical Recipes is an excellent introduction to the topic, notably chapters 2 and 11.

Solving the eigenvalue problem efficiently is too complicated for this course, but we will explain one of the basic methods for solving a system of linear equations. However, we will not write even this method, but instead rely on an external numerical package.

1.2.1 Finding eigenvalues

Linear algebra is one of the best excuses for using an external numerical code repository. Here we'll use a couple of externally provided routines. The eigenvalue routine comes from Numerical Recipes, second edition.

One of the basic tasks that any numerical package has to handle is how to define matrices. As we mentioned before, C does not handle two-dimensional arrays gracefully. For simplicity, however, we will stick with the basic implementation, namely to define matrices using

```
#define NDIM 6
double mymatrix[NDIM][NDIM];
```

This creates a two-dimensional array with elements 0.5 in each dimension.

As we discussed previously, the major disadvantage of this implementation is that when passing the matrix to a function (as we hope you will often do), you need to include the size of the array *in your source code*. In other words, you write

```
void myfunction(double mymatrix[NDIM][NDIM]);
```

and call as `myfunction(mymatrix)`. This means that the size of the matrix is not adjustable at run-time, nor can the function be called with two differently sized matrices in the same program. This is a fatal flaw for any serious work, but we can live with it for our two weeks¹.

Once you have your matrices, you can compute the eigenvalues of a symmetric real-valued matrix with function

```
double mymatrix[NDIM][NDIM], eigenvectors[NDIM][NDIM], eigenvalues[NDIM];
// The above is to define variables; you have to then load them.
// Then call the routine:
eigensym(mymatrix, eigenvectors, eigenvalues, NDIM);
```

This will destroy the values in `mymatrix` (make a copy if you want to keep them!) and put the eigenvalues into the vector `eigenvalues` and put the corresponding eigenvectors into the columns of the matrix `eigenvectors`. To be clear, the i^{th} eigenvalue is in `eigenvalues[i]` and its eigenvector is in `eigenvectors[0..N-1][i]`. N is the dimension of the space.

¹There are alternatives, but you have to either set up arrays of pointers or bypass the `[] []` notation. Many of the recent numerical packages have been written in object-oriented languages like C++ in large part to avoid this style of matrix definition.

Remember that `mymatrix` must be a symmetric matrix!

You will need the file `/home/doug/phys305/linalg305b.c`. Include this file in your compile lines, and put the prototype

```
eigensym(double mymatrix[][] , double evec[][], double eval[], int dim);
```

for the relevant functions at the top of your files.

1.3 Solving Linear Systems of Equations

One frequently runs into the need to solve a system of *linear equations*, equations in which the unknowns appear in only linear combinations. For a simple system of two equations in two unknowns (a “two-by-two system” or a “system of order two”) we can easily write down the algebraic solution. For larger systems this becomes both tedious and subject to serious problems with round-off error. Very large systems are frequently encountered in numerical simulations, with systems of order 10^6 or even larger quite common. Clearly for these, a computer approach is required.

The subject of linear system solution is quite large, with many techniques having been developed over the years to deal with special problems or systems with special properties that are frequently encountered. In these notes we will consider only the most basic and most frequently used: Gaussian Elimination. *Numerical Recipes* has a very clear and fairly complete treatment in chapter two. Another method, *LU Decomposition*, is perhaps a better choice in general, but its theory is beyond the level of this course.

1.3.1 Elimination

To solve a three by three system

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

one might begin (assuming that $a_{11} \neq 0$) by solving the first equation for x_1

$$x_1 = \frac{b_1 - a_{12}x_2 - a_{13}x_3}{a_{11}}$$

and then substituting this solution into the system to eliminate the variable x_1 and end up with a two by two system:

$$\begin{aligned} a'_{22}x_2 + a'_{23}x_3 &= b'_2 \\ a'_{32}x_2 + a'_{33}x_3 &= b'_3 \end{aligned}$$

where the coefficients would now be

$$\begin{aligned} a'_{22} &= a_{22} - \frac{a_{21}}{a_{11}}a_{12} & a'_{23} &= a_{23} - \frac{a_{21}}{a_{11}}a_{13} & b'_2 &= b_2 - \frac{a_{21}}{a_{11}}b_1 \\ a'_{32} &= a_{32} - \frac{a_{31}}{a_{11}}a_{12} & a'_{33} &= a_{33} - \frac{a_{31}}{a_{11}}a_{13} & b'_3 &= b_3 - \frac{a_{31}}{a_{11}}b_1 \end{aligned}$$

We might write these new coefficients more generally as

$$\begin{aligned} a'_{ij} &= a_{ij} - \frac{a_{i1}}{a_{11}}a_{1j} \\ b'_i &= b_i - \frac{a_{i1}}{a_{11}}b_1 \end{aligned}$$

Repeating the process for the two by two system, we might solve for x_2 (assuming that $a'_{22} \neq 0$) and substitute that into the second equation to obtain

$$a''_{33}x_3 = b''_3$$

where

$$\begin{aligned} a''_{33} &= a'_{33} - \frac{a'_{32}}{a'_{22}}a'_{23} \\ b''_3 &= b'_3 - \frac{a'_{32}}{a'_{22}}b'_2 \end{aligned}$$

or more generally as

$$\begin{aligned} a''_{ij} &= a'_{ij} - \frac{a'_{i2}}{a'_{22}}a'_{2j} \\ b''_i &= b'_i - \frac{a'_{i2}}{a'_{22}}b'_2. \end{aligned}$$

We have reduced the original system into one of the form

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 &= b_1 \\ a'_{22}x_2 + a'_{23}x_3 &= b'_2 \\ a''_{33}x_3 &= b''_3. \end{aligned}$$

Such a system has a coefficient matrix which has zeros below its diagonal elements – it is said to be of *upper triangular form*. As we considered each row i in turn, we used it to eliminate the coefficients in column i for all rows beneath it. This forward sweep through the equations is known as *forward elimination*.

Upper triangular form is very convenient in determining the unknowns. The last row is just the solution for the last unknown. Each previous row contains its own unknowns and only those others which lie in rows below it. To solve the system, in each row we just plug in the known x_i from the lower rows and solve for the remaining one. In our example, the last row tells us that $x_3 = b''_3/a''_{33}$. We can substitute this

back into the second equation to determine $x_2 = (b'_2 - a'_{23}x_3)/a'_{22}$, and then substitute that in turn back into the first equation to determine $x_1 = (b_1 - a_{12}x_2 - a_{13}x_3)/a_{11}$. The sweep backward through the triangularized equations is known as *back substitution*.

Together, forward elimination and back substitution in this manner are called *Gaussian Elimination* and is our first method of solving a linear system. We might write an algorithm for it as:

```

! forward elimination

for i = 1 to N                ! consider each row in turn

    for j = i+1 to N          ! eliminate unknown i from
                              ! subsequent rows

        fac = a_{j,i}/a_{i,i}
        for k = 1 to n
            a_{j,k} = a_{j,k} - fac * a_{i,k}
        end k loop

        b_{j} = b_{j} - fac * b_i
    end j loop
end i loop

! back substitution

b_N = b_N/a_{N,N}            ! start with last equation

for i = N-1 to 1 step -1     ! move back up the system
    for j = i+1 to N
        x_i = b_i - a_{i,j} * x_j
    end j loop
end i loop

```

The astute reader will find two problems with this procedure. The first is what to do when a coefficient on the diagonal is zero, since we cannot divide by it. The second problem is that there are a lot of subtractions, and this leads to round-off error. In practice, round-off error renders the whole Gaussian Elimination process nearly useless!

There is a way around these problems, known as *pivoting*. At each step (at the beginning of the overall i -loop above), we rearrange our remaining equations so that the element we divide by is the largest one in that column. It turns out that this solves both the divide-by-zero and the round-off problems at once. One should never use Gaussian elimination without this pivoting.

As the elimination progresses, if there are no non-zero elements in a column of the remaining equations, we have a *singular* system with no solution; we will have more to say on this below.

Let us write an algorithm for Gaussian Elimination with pivoting. To avoid having to constantly reshuffle the equations, we will employ a trick known as a *pointer*: an array of values which point to other values. Instead of rearranging the rows, we will keep an array of integers `indx(i)` which tells us which equation we want in the i -th row of our rearranged solution. If we always refer to row i as `indx(i)`, then all we have to do to swap the positions of two rows is to swap the contents of two elements of `indx`.

Having written a Gaussian Elimination routine once, solving any linear system is easy. You merely fill up a matrix of coefficients a_{ij} and a right hand side b_i , hand them to the routine, and you get back the solution. Here, then, is our algorithm for Gaussian Elimination with pivoting. Like many such algorithms, to avoid needing duplicate storage, this one destroys the original matrix and replaces the right hand side with the solution when it returns.

```

routine Gauss (a, n, b)

for i = 1 to N                                ! initially, our equations are
  indx(i) = i                                  ! in order
end i loop

! forward elimination

for i = 1 to N                                ! consider each row in turn

  ! find the row to use for pivoting, i.e. the row with the largest
  ! magnitude element in column i
  pmax = 0
  for j = i to N
    if abs(a(indx(j),i)) > pmax then {
      jmax = j
      pmax = abs(a(indx(j),i))
    }
  end j loop

  ! swap this row with row i
  k = indx(i)
  indx(i) = indx(jmax)
  indx(jmax) = k

```

```

! check to see this largest element is not zero
diag = a(indx(i),i)
if diag = 0 then {
  write "singular matrix"
  quit
}

! scale the pivot row to have unit diagonal
for j = 1 to N
  a(indx(i),j) = a(indx(i),j) / diag
end j loop
b(indx(i)) = b(indx(i)) / diag

! eliminate unknown indx(i) from subsequent rows
for j = i+1 to N
  fac = a(indx(j),i)
  for k = i+1 to N
    a(indx(j),k) = a(indx(j),k) - fac * a(indx(i),k)
  end k loop
  b(indx(j)) = b(indx(j)) - fac * b(indx(i))
end j loop
end i loop

! back substitution

b(indx(n)) = b(indx(n))/a(indx(n),n)    ! start with the last equation

for i = N-1 to 1 step -1    ! move back up the system
  for j = i+1 to N
    b(indx(i)) = b(indx(i)) - a(indx(i),j) * b(indx(j))
  end j loop
end i loop

! reorder the solution to correspond to the original order as given
for i = 1 to N
  a(i,1) = b(indx(i))! use a(i,1) as temporary storage
end i loop
for i = 1 to N
  b(i) = a(i,1)
end i loop

```

1.3.2 Numerical Implementation

Again, we will provide you with a code, written by J. Richardson, to do this. It is also in `/home/doug/phys305/linalg305b.c`. Two routines are provided:

```
matinv(double mymatrix[] [], double matrix_inverse[] [], int dim);
matinv(double mymatrix[] [], double vector[], double answer[], int dim);
```

The first one inverts `mymatrix`, putting the result in `matrix_inverse`. The second solves $\mathbf{A}\vec{x} = \vec{b}$ where \vec{b} is in `vector` and \vec{x} is in `answer`. In both cases, `dim` is the dimension N of the array and the matrix `mymatrix` is returned in a scrambled state.

Put these prototypes at the top of your file and then include the `linalg305b.c` file in the compile line or Makefile.

1.3.3 Troubles

All this seems simple, and it is if the matrix of coefficients is well-behaved. You merely set up the equations by filling in the appropriate elements of \mathbf{A} with the coefficients and \vec{b} with the right hand sides, and then call the linear system routine to obtain a solution.

There is a class of matrices, however, for which there *is no solution*. These matrices are known as *singular* matrices. Define the *inverse* of the matrix \mathbf{A} , denoted \mathbf{A}^{-1} , by the relation

$$\mathbf{A}^{-1}\mathbf{A} = \mathbf{1}$$

where $\mathbf{1}$ is the identity matrix, with one on the diagonal and zeros for all off-diagonal elements.

The solution to a linear system can then be written symbolically as

$$\vec{x} = \mathbf{A}^{-1}\vec{b}.$$

It is important to note that solving a linear system does not take as much computational effort as inverting (determining the inverse of) a matrix; we shall show below that determining the inverse of a matrix is the same as solving N linear systems, where N is the order of the system.

A singular matrix has no inverse. Thus, we cannot write the solution symbolically as $\vec{x} = \mathbf{A}^{-1}\vec{b}$, and indeed a linear system whose matrix of coefficients is singular has no solution. This corresponds, roughly, to the scalar equation $ax = b$ with $a = 0$; the equation does not determine the value of x . Likewise, if the system of equations with coefficient matrix \mathbf{A} does not determine the solution of the system, \mathbf{A} is said to be singular.

Take as an example a linear system of 3 equations in 3 unknowns for which one of the equations is a multiple of one of the others. In this case, we really have only 2

linearly independent equations. For example, in the system of equations

$$\begin{aligned} 2x + 3y - z &= 2 \\ 4x + 6y - 2z &= 4 \\ x + y + z &= 1 \end{aligned}$$

the second equation is twice the first, so this second equation does not give us any additional information about the solution, and the matrix of coefficients

$$\begin{pmatrix} 2 & 3 & -1 \\ 4 & 6 & -2 \\ 1 & 1 & 1 \end{pmatrix}$$

is singular, though at first glance there is nothing particularly odd-looking about it.

If you run into a linear system with a singular matrix it probably means that you made a conceptual error in setting up the system of equations; you haven't really provided N independent constraints upon the values of the N unknowns you wish to determine. Life in the finite-precision world of numerical work is occasionally more cruel than this, however. It is possible to write down a system of equations which, while perfectly well-determined in infinite precision arithmetic, does not admit of a solution in finite precision. Such a system is said to be *numerically singular*.

For example, take the following (somewhat absurd) system of order 2:

$$\begin{aligned} x + y &= 2 \\ 10^{18}x + (10^{18} - 1)y &= 3 \end{aligned}$$

The solution is $x = 5 - 2 \times 10^{18}$, $y = -3 + 2 \times 10^{18}$. Note, however, that the second equation is *very nearly* 10^{18} times the first; indeed, to the 16 significant figure accuracy we can obtain in double precision, the second equation is *exactly* such a multiple, and the matrix is numerically singular in double precision arithmetic. Even though we can write down the correct solution to this system quite simply, we cannot solve the system by the straightforward application of Gaussian Elimination.

Even more insidious is a matrix which is *nearly* singular. Take, for example, the linear system $\mathbf{A}\vec{x} = \vec{b}$ which has the coefficient matrix $A_{ij} = 1/(i + j)$ (the i -th row, j -th column of \mathbf{A} has the value $1/(i + j)$), and $b_i = i$ (the right hand side of the i -th equation is i). This is a perfectly nice-looking matrix, and none of the rows is a multiple of any other.

For this example, write a simple program to solve this linear system for various orders (i.e. various numbers of variables), using double precision and Gaussian Elimination. As a check on the accuracy of the solution, multiply the original matrix by the solution determined, and then compute the relative error between the resulting right hand side and the original. In other words, determine a solution \vec{x} , and compute $\vec{b}' = \mathbf{A}\vec{x}$, and compute the maximum relative error in the components of \vec{b}' : $\epsilon = \max_i [(b'_i - b_i)/b_i]$. Here is the result:

order	relative error	# of significant figures
2	8.882E-16	15.1
4	1.525E-14	13.8
8	9.720E-10	9.0
16	8.125E-07	6.1
32	1.745E-05	4.8
64	7.919E-03	2.1
128	1.964E-03	2.7
256	1.327E-03	2.9
512	9.023e-02	1.0

While for the 2-by-2 system you get 15 significant figures of accuracy, nearly what could be expected as the best case from 16 significant figure arithmetic, for the 64-by-64 case you get only 2 significant figures, and for the 512-by-512 case, only one – practically garbage!

In this case, \mathbf{A} is nearly singular, and becomes more nearly singular as the order of the matrix is increased. Thus, while this is a well-posed problem mathematically, it becomes increasingly ill-posed in finite precision.

The moral of this story is of course that one should never assume that a numerical solution of a linear system is exact; if you are at all unsure of your result, check on the solution's accuracy; comparing right hand sides is just one (very easy to perform) way of doing so.

The LU decomposition method (e.g., `ludcmp` from Numerical Recipes) will tell you if your matrix is (either numerically or exactly) singular by issuing an error message; it will not, however, tell you if your solution is approaching pure nonsense!

There is a way to deal, approximately, with nearly-singular systems, based on another matrix decomposition, called *singular value decomposition*. It is beyond the scope of this course, but be aware of its existence if you run into such problems.

The second problem mentioned above is the determination of the inverse of the matrix \mathbf{A} , \mathbf{A}^{-1} . We write down the identity

$$\mathbf{A}\mathbf{A}^{-1} = \mathbf{1};$$

from this, you can see that this is really N linear systems of equations. The i -th column of \mathbf{A}^{-1} is the solution to the system with a right hand side which is all zeros except for the i -th component which is unity.

Thus, we can determine the inverse of a matrix by solving N linear systems for the columns of \mathbf{A}^{-1} , with the same matrix \mathbf{A} and different right hand sides. Using *LU* decomposition, we can determine the decomposition of \mathbf{A} once, and then back substitute N times. From this, you can see that it takes less time to solve a linear system than to determine the inverse of the matrix of coefficients and then multiply the right hand side.

For our application to maximum likelihood, it is important to note that the matrix α is often nearly singular. This occurs when the data do not clearly distinguish between two or more of the basis functions provided. The equations are both over-determined, in the sense of having more equations than unknowns, and under-determined, in the sense of having two or more (nearly) indistinguishable solutions. In the case of an over-determined system, Singular Value Decomposition (SVD) gives a solution which is the best approximation in the least-squared sense, which is quite appropriate here! SVD is beyond the scope of this course, but if you find that you cannot get a good solution using LU decomposition, have a look at *Numerical Recipes*, which gives a very nice description of the method as well as its application in the present context.