Linear Algebra Review
Daniel Kersten

Vector operations

■ **Dimension of a vector.**

You can get the dimensionality of a vector using **Dimensions[]**, or **Length[]**.

```
v = {2.1, 3, -0.45, 4.9};

Dimensions[v]
```

```
{4}
```

Dimensions[], will give you the dimensions of a matrix, while **Length[]** tells you the number of elements in the list. For example,

```
M = {{2,4,2}, {1,6,4}};
```

```
Length[M]
```

```
2
```

Try comparing **Length[M]** with **Dimensions[M]**.

■ **Transpose of a vector.**

The transpose of a column vector is just the same vector arranged in a row. However, because of the way Mathematica uses lists to represent vectors you don't have to distinguish between row and column vectors. In standard math notation, transpose of a vector **x**, is often written $\mathbf{x}^{\mathbf{T}}$. You can see a vector in column form by typing **v//MatrixForm**, or**:**


```
MatrixForm[v]
```

$$\begin{pmatrix} 2.1 \\ 3 \\ -0.45 \\ 4.9 \end{pmatrix}$$

■ **Vector addition** is accomplished by simply adding the components of each vector to make a new vector. Note that the vectors all have the same dimension.

```
a = {3,1,2};
b = {2,4,8};
c = a + b
```

```
{5, 5, 10}
```

Vectors can be multiplied by a constant. We saw an example of this earlier.

```
2 a
```

```
{6, 2, 4}
```

■ **Metric length of a vector.**

It is unfortunate terminology, but **Length[]** does NOT give you the metrical length of the vector. In order to get the length of a vector, you calculate the Euclidean distance from the origin to the end-point of the vector. We get this by squaring each component, adding up the squares, and taking the square root. First, we will do this using the **Apply[]** function, where the **Plus** operation is applied to all the elements of the list. Note that the operation of exponentiation is "listable", that is it is applied to each element of the vector:

```
a^2
```

```
{9, 1, 4}
```

What is **a a** ?

```
N[Sqrt[Apply[Plus, a^2]]]
```

```
3.74166
```

If you wish, you can define your own function to apply to the list. What we have just calculated is the square root of the dot product or inner product of **a** with itself. The length of a vector **a** is often written as **|a|** in standard math notation. In the next section, we use the inner or dot product to calculate the metric length of a vector.

■ **Inner product**. To calculate the inner product of two vectors, you multiply the corresponding components and add them up:

```
u = {u1,u2,u3,u4};
v = {v1,v2,v3,v4};
u.v
```

```
u1 v1 + u2 v2 + u3 v3 + u4 v4
```

The **inner product** is also called the **dot product**. Later we will see what is meant by **outer product**. The inner product between two vectors **a** and **b** is written either as:

a.b or **[a,b]**, or $\mathbf{a}^T\mathbf{b}$

Mathematica uses the dot notation.

One use of the inner product is to calculate the length of a vector. **a.a** is just the sum of the squares of the elements of **a**, so gives us another way of calculating the length of a vector.

```
N[Sqrt[a.a]]
```

```
3.74166
```

Let's define a function that will return the length of a vector, x:

```
Vectorlength[x_] := N[Sqrt[x.x]]
```

■ **Projection.** The dot product, **a.b**, is equal to:

|a| |b| cos(angle between **a** and **b**)

In problem set 1, you calculate the output of a linear neuron model as the dot product between an input vector and a weight vector. Both the weight and input lists can be thought of as vectors in an n-dimensional space. Suppose the weight vector has unit length. Recall that you can normalize any vector to unit length by dividing by its length:

```
v = v/Sqrt[v.v] ;
```

Geometrically, we can think of the output of a neuron as the projection of the activity of the neuron input activity vector onto the weight vector direction. Suppose the input vector is already perpendicular to the weight vector, then the output of the neuron is zero, because the cosine of 90 degrees is zero. As you found or will find with the cross-correlator of Problem Set 1, the further the input pattern is away from the weight vector, as measured by the cosine between them, the poorer the

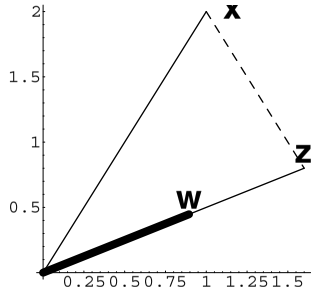
"match" between input and weight vectors, and the lower the response.

Here are three lines of code that calculate the two-dimensional vector **z** in the direction of **w**, with a length determined by "how much of **x** projects in the **w** direction":

```
x = {1,2};
w = N[{2/Sqrt[5],1/Sqrt[5]}];
z = (x.w) w ;
```

```
Show[   Graphics[{Line[{{0,0}, x}],
            Line[{{0,0}, z}],
            {Dashing[{0.03,0.03}], Line[{x, z}] },
            Text[FontForm["w",{"Helvetica-Bold",18}], w, {0,-1}],
            Text[FontForm["x",{"Helvetica-Bold",18}], x, {-2,0}],
            Text[FontForm["z",{"Helvetica-Bold",18}], z, {0,-1}],
            {AbsoluteThickness[3], Line[{{0,0}, w}] }
        }],
        Axes->True, AspectRatio->1

];
```



■ **Angle between two vectors and orthogonality: Similarity measure between patterns**

Often we will want some measure of the similarity between two patterns of neural firings. As we have just seen, one measure of comparison is the degree to which the two state vectors point in the same direction. The cosine of the angle between two vectors is one possible measure:

```
cosine[x_,y_] := x.y/(Vectorlength[x] Vectorlength[y])
cosine[a,b]
```

```
0.758175
```

Note that if two vectors point in the same direction, the cosine of the angle between them is 1:

```
a = {2,1,3,6};
b = {6, 3, 9, 18};
cosine[a,b]
```

```
1.
```

Try verifying that **w** and **z** from the previous section point in the same direction.

If two vectors point in the opposite directions, the cosine of the angle between them is -1:

```
a = {-2,-1,-3,-6};
b = {6, 3, 9, 18};
cosine[a,b]
```

```
−1.
```

Two vectors may point in the same direction, but could be quite different because they have different lengths. Another measure of similarity is the length of the difference between two vectors:

```
Vectorlength[a - b]
```

```
28.2843
```

■ **Orthogonality.** The case where vectors are at right angles to each other is an important special case that is worth spending a little time on. Consider an 8-dimensional space. One very familiar set of orthogonal vectors is the following:

```
u1 = {1,0,0,0,0,0,0,0};
u2 = {0,1,0,0,0,0,0,0};
u3 = {0,0,1,0,0,0,0,0};
u4 = {0,0,0,1,0,0,0,0};
u5 = {0,0,0,0,1,0,0,0};
u6 = {0,0,0,0,0,1,0,0};
u7 = {0,0,0,0,0,0,1,0};
u8 = {0,0,0,0,0,0,0,1};
```

Each vector has unit length, and it is easy to see just by inspection that the inner product between any two is zero. On the other hand, here is another set of 8 vectors in 8-space for which it is not immediately obvious that they are all orthogonal. These vectors are called Walsh functions:


```
v1 = {1, 1, 1, 1, 1, 1, 1, 1};
v2 = {1,-1,-1, 1, 1,-1,-1, 1};
v3 = {1, 1,-1,-1,-1,-1, 1, 1};
v4 = {1,-1, 1,-1,-1, 1,-1, 1};
v5 = {1, 1, 1, 1,-1,-1,-1,-1};
v6 = {1,-1,-1, 1,-1, 1, 1,-1};
v7 = {1, 1,-1,-1, 1, 1,-1,-1};
v8 = {1,-1, 1,-1, 1,-1, 1,-1};
```

You can calculate the inner products between any two, and you will find out that they are all zero. Note that with the first set of vectors, $\{\mathbf{u}_i\}$, you can tell which vector it is just by looking for where the 1 is. For the second set, $\{\mathbf{v}_i\}$, you can't tell by looking at just one component. For example, the first component of all of the Walsh functions has a 1. You have to look at the pattern to tell which Walsh function you are looking at.

Suppose for the moment that we want to assign meaning to each of the patterns--each pattern is a code for some thing, like "grandma Tompkins", "grandma Wilke", and so forth. If we use the **u**'s, then we could look for the one neuron that lights up to find out which grandma it is representing--then neuron activity represented, for example, by the third element of the pattern could mean "grandma Wilke". This strategy wouldn't work if we encoded a collection of grandmothers using the **v**'s. The **v**'s give us a simple example of what is sometimes referred to as a **distributed code**. The **w**'s are examples of a **grandmother cell code**. The reason for this obscure terminology can be traced to earlier debates on whether there may be single cells in the brain whose firing uniquely determines the recognition of one's grandmother.

■ **Orthonormality.** The Walsh set is orthogonal, but they are not of unit length. We have already seen some of the advantages of working with unit length vectors. The general issue of normalization comes up all the time in neural networks both in terms of limiting overall neural activity, and limiting synaptic weights. So it is sometimes convenient to normalize an orthogonal set, producing what is known as an *orthonormal* set of vectors:

```
w1 = v1/Vectorlength[v1];
w2 = v2/Vectorlength[v2];
w3 = v3/Vectorlength[v3];
w4 = v4/Vectorlength[v4];
w5 = v5/Vectorlength[v5];
w6 = v6/Vectorlength[v6];
w7 = v7/Vectorlength[v7];
w8 = v8/Vectorlength[v8];
```

Vector representations, linear algebra

The issue of how information is to be represented is fundamental in the information sciences generally, as well as for neural network theory. A pattern of activity over a set of neurons is presumed to mean something, and there are different ways of coding the same meaning. But different codes have different properties. A code may not be sufficient to uniquely code all the possible things we need to represent. A code could be redundant and have more than one way of representing the same thing. This section continues with our review of the basics of vector and linear algebra by going a little more deeply into the subject. The pay-off will be some mathematics that provides intuition about issues of neural representation. You can think of this as a first lesson in the "psychology of linear algebra".

■ Basis sets

It is pretty clear that given any vector whatsoever in 8-space, you can specify how much of it gets projected in each of the eight directions specified by the unit vectors v1, v2, ...v8. But you can also build back up an arbitrary vector by adding up all the contributions from each of the component vectors. This is a consequence of vector addition and can be easily seen to be true in 2 dimensions. We can verify it ourselves. Pick an arbitrary vector g, project it onto each of the basis vectors, and then add them back up again:

```
g = {2,6,1,7,11,4,13, 29} ;
```

```
(g.u1) u1  + (g.u2) u2  +(g.u3) u3  +(g.u4) u4  +
(g.u5) u5  +(g.u6) u6  +(g.u7) u7  +(g.u8) u8
```

```
{2, 6, 1, 7, 11, 4, 13, 29}
```

■ Exercise

What happens if you project **g** onto the normalized Walsh basis set defined by **{w1,w2,...}** above, and then add up all 8 components?

```
(g.w1) w1  + (g.w2) w2  +(g.w3) w3  +(g.w4) w4  +
(g.w5) w5  +(g.w6) w6  +(g.w7) w7  +(g.w8) w8
```

The projections, $\mathbf{g.u_i}$ are sometimes called the **spectrum** of **g**. This terminology comes from the Fourier basis set used in Fourier analysis. A discrete version of a Fourier basis set is similar to the Walsh set, except that the elements fit a sine wave pattern, and so are not binary-valued.

The orthonormal set of vectors we've defined above is said to be **complete**, because any vector in 8-space can be expressed as a linear weighted sum of these **basis vectors**. The weights are just the projections. If we had only 7 vectors in our set, then we would not be able to express any 8-dimensional vector in terms of this basis set. The seven vector set would be said to be **incomplete**. A basis set which is orthonormal and complete is very nice from a mathematical point of view. Another bit of terminology is that these seven vectors would not **span** the 8-dimensional space. But they would span some sub-

space, that is of smaller dimension, of the 8-space.

There has been much interest in describing the effective weighting properties of visual neurons in primary visual cortex of higher level mammals (cats, monkeys) in terms of basis vectors. One issue is if the input (e.g. an image) is projected (via a collection of receptive fields) onto a set of neurons, is information lost? If the set of weights representing the receptive fields of the collection of neurons is complete, then no information is lost.

■ Linear dependence

What if we had 9 vectors in our basis set used to represent vectors in 8-space? For the u's, it is easy to see that in a sense we have too many, because we could express the 9th in terms of a sum of the others. This set of nine vectors would be said to be linearly dependent. A set of vectors is linearly dependent if one or more of them can be expressed as a linear combination of some of the others. Sometimes there is an advantage to having an "over-complete" basis set (e.g. more than 8 vectors for 8-space; cf. Simoncelli et al., 1992).

Theorem: A set of mutually orthogonal vectors is linearly independent.

However, note it is quite possible to have a linearly independent set of vectors which are not orthogonal to each other. Imagine 3-space and 3 vectors which do not jointly lie on a plane. This set is linearly independent.

If we have a linearly independent set, say of 8 vectors for our 8-space, then no member can be dropped without a loss in the dimensionality of the space spanned.

It is useful to think about the meaning of linear independence in terms of geometry. A set of three linearly independent vectors can completely span 3-space. So any vector in 3-space can be represented as a weighted sum of these 3. If one of the members in our set of three can be expressed in terms of the other two, the set is not linearly independent and the set only spans a 2-dimensional subspace. That is, the set can only represent vectors which lay on a plane in 3-space. This can be easily seen to be true for the set of u's, but is also true for the set of v's.

■ Thought exercise

Suppose there are three inputs feeding into three neurons in a simple linear network. If the weight vectors of the three neurons are not linearly independent, do we lose information?

Basic matrix arithmetic

Definition of a matrix: a list of scalar lists

An mxn matrix has m rows, and n columns. Here is a 3x4 matrix:

```
MatrixForm[
    Table[W[i,j],{i,1,3},{j,1,4}]]
```

$$\begin{pmatrix} W(1,1) & W(1,2) & W(1,3) & W(1,4) \\ W(2,1) & W(2,2) & W(2,3) & W(2,4) \\ W(3,1) & W(3,2) & W(3,3) & W(3,4) \end{pmatrix}$$

The matrix can be written in standard form using subscripts as:

$$\begin{pmatrix} w_{11} w_{12} w_{13} w_{14} \\ w_{21} w_{22} w_{23} w_{24} \\ w_{31} w_{32} w_{33} w_{34} \end{pmatrix}$$

And as we have seen, *Mathematica* represents a matrix as a list of lists:

```
%//StandardForm
```

```
{{W[1, 1], W[1, 2], W[1, 3], W[1, 4]},
 {W[2, 1], W[2, 2], W[2, 3], W[2, 4]}, {W[3, 1], W[3, 2], W[3, 3], W[3, 4]}}
```

Remember, the symbol % in *Mathematica* stands for the most recent output, %% stands for the second to last output, and so forth.

Adding, subtracting and multiplying by a scalar

As with vectors, matrices are added, subtracted, and multiplied by a scalar component by component:

```
A = {{a,b},{c,d}};
B = {{x,y}, {u,v}};
```

Let's add **A** to **B**:

```
A+B
```

$$\begin{pmatrix} a+x & b+y \\ c+u & d+v \end{pmatrix}$$

Subtracting **B** from **A**:


```
A-B
```

$$\begin{pmatrix} a-x & b-y \\ c-u & d-v \end{pmatrix}$$

And if we multiply **A** by 3:

```
3 A
```

$$\begin{pmatrix} 3a & 3b \\ 3c & 3d \end{pmatrix}$$

Multiplying two matrices

We have already seen how to multiply a vector by a matrix: we replace the i^{th} row of the output vector by the inner product of the i^{th} row of the matrix with the vector.

In order to multiply a matrix **A**, by another matrix **B** to get **C = AB**, we calculate the ij^{th} component of the output matrix by taking the inner product of the i^{th} row of **A** with the j^{th} column of **B**:

```
A.B
```

$$\begin{pmatrix} bu+ax & bv+ay \\ du+cx & dv+cy \end{pmatrix}$$

Note that **AB** is not necessarily equal to **BA**:

```
B.A
```

$$\begin{pmatrix} ax+cy & bx+dy \\ au+cv & bu+dv \end{pmatrix}$$

Laws of commutation, association and distribution

Look at the element in the upper left of the matrix **BA** above--there is no reason, in general, for **ax+bu** to equal **ax + cy**. That is, matrix multiplication does not *commute*.

Apart from commutation for matrix multiplication, the usual laws of commutation, association, and distribution that hold for scalars hold for matrices. Matrix addition and subtraction do commute. Matrix multiplication is associative, so **(AB)C = A(BC)**. The distributive law works too:

A(B+C) = AB + AC

Non-square matrices

It is not necessary for **A** and **B** to be square matrices (i.e. have the same number of rows as columns) to multiply them. But if **A** is an mxn matrix, then **B** has to be an nxp matrix in order for **AB** to make sense. For example, here **F** is a 3x2 matrix, and **G** is a 2x4 matrix.

```
F = {{a,b},{c,d},{e,f}};
G = {{p,q,r,s},{t,u,v,w}};
```

```
Dimensions[F]
```

```
{3, 2}
```

```
Dimensions[G]
```

```
{2, 4}
```

Because **F** has 2 columns, and **G** has 2 rows, it makes sense to multiply **G** by **F**:

```
F.G
```

$$\begin{pmatrix} a\,p+b\,t & a\,q+b\,u & a\,r+b\,v & a\,s+b\,w \\ c\,p+d\,t & c\,q+d\,u & c\,r+d\,v & c\,s+d\,w \\ e\,p+f\,t & e\,q+f\,u & e\,r+f\,v & e\,s+f\,w \end{pmatrix}$$

However, because the number of columns of **G** (4) do not match the number of rows of **F** (3), **G.F** is not well-defined:


```
G.F
```

Dot::dotsh : Tensors $\begin{pmatrix} p & q & r & s \\ t & u & v & w \end{pmatrix}$ and $\begin{pmatrix} a & b \\ c & d \\ e & f \end{pmatrix}$ have incompatible shapes.

$$\begin{pmatrix} p & q & r & s \\ t & u & v & w \end{pmatrix}.\begin{pmatrix} a & b \\ c & d \\ e & f \end{pmatrix}$$

Inverse of a Matrix

Dividing a matrix by a matrix: the identity matrix & matrix inverses

The matrix corresponding to 1 or unity is the **identity matrix**. Like 1, the identity matrix is fundamental enough, that *Mathematica* provides a special function to generate n-dimensional identity matrices. Here is a 2x2:

```
IdentityMatrix[2]
```

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

It is easy to show that the identity matrix plays the role for matrix arithmetic that the scalar 1 plays for scalar arithmetic.

How can one divide one matrix, say **B**, by another, say **A**? We can divide numbers, x by y, by multiplying x times the inverse of y, i.e. 1/y. So to do the equivalent of dividing **B** by **A**, we need to find a matrix **Q** such that when **A** is multiplied by **Q**, we get the matrix equivalent of unity, i.e. the identity matrix. Then "**B/A**" can be achieved by calculating the matrix product: **B.Q**.

```
A = {{a,b},{c,d}};
```

Mathematica provides a built-in function to compute matrix inverses:

```
Q = Inverse[A]
```

$$\begin{pmatrix} \frac{d}{a\,d-b\,c} & -\frac{b}{a\,d-b\,c} \\ -\frac{c}{a\,d-b\,c} & \frac{a}{a\,d-b\,c} \end{pmatrix}$$

We can test to see whether the product of a A and Q is the identity matrix, but *Mathematica* won't go through the work of simplifying the algebra in this case, unless we specifically ask it to.

```
Q.A
```

$$\begin{pmatrix} \frac{a\,d}{a\,d-b\,c}-\frac{b\,c}{a\,d-b\,c} & 0 \\ 0 & \frac{a\,d}{a\,d-b\,c}-\frac{b\,c}{a\,d-b\,c} \end{pmatrix}$$

```
Simplify[Q.A]
```

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Here is a simple numerical example:

```
B = {{1,-1},{3,2}};
R = Inverse[B]
```

$$\begin{pmatrix} \frac{2}{5} & \frac{1}{5} \\ -\frac{3}{5} & \frac{1}{5} \end{pmatrix}$$

```
B.R
```

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

Badly conditioned matrices

What if one row is a scaled version of another row? Then the rows are not linearly independent. In this case, the inverse is not defined.

```
B1 = {{1.5,1},{3,2.0}}
```

$$\begin{pmatrix} 1.5 & 1 \\ 3 & 2. \end{pmatrix}$$